PROYECTO FIN DE CARRERA

CURSO 2012-2013

VERIFICACIÓN DE SEGURIDAD DE UN PROTOCOLO CRIPTOGRÁFICO UTILIZANDO UN ASISTENTE DE DEMOSTRACIÓN

Autores:

Jorge Miguel Gutiérrez-Barquín

Luis Alberto Inga Rivera

Directores:

Ricardo Peña Marí, Dep.Sistemas informáticos y Computación

María Emilia Alonso, Dep.Algebra

Este trabajo se lo dedicamos

a nuestra familia más cercana.

AUTORIZACIÓN

Jorge Miguel Gutiérrez-Barquín y Luis Alberto Inga Rivera, alumnos matriculados en la asignatura de Sistemas Informáticos, autorizan, mediante el presente documento, a la Universidad Complutense de Madrid (UCM), a difundir y utilizar con fines académicos, no comerciales, y mencionando expresamente a sus autores, tanto la propia memoria, como el código y la documentación , todo ello realizado durante el curso académico 2012-2013 bajo la dirección de Ricardo Peña Marí, profesor del Departamento de Sistemas Informáticos y Computación de la Facultad de Informática de dicho organismo.

Jorge Miguel Gutiérrez-Barquín Luis Alberto Inga Rivera

AGRADECIMIENTOS

Agradecemos a nuestras familias por habernos apoyado a lo largo de toda la carrera, cuyo final está cada vez más cerca y culmina con la presentación del proyecto de Sistemas Informáticos.

No sería justo olvidarnos de nuestros directores de proyecto, Ricardo Peña Marí y María Emilia Alonso por prestarnos su ayuda durante todo el proceso y haber aportado su granito de arena.

RESUMEN

El objetivo de este proyecto denominado"VERIFICACIÓN DE SEGURIDAD DE UN PROTOCOLO CRIPTOGRAFICO UTILIZANDO UN ASISTENTE DE DEMOSTRACIÓN" es demostrar usando el asistente de demostración Isabelle, que el protocolo criptográfico de conocimiento cero llamado de Fiat-Shamir es un ∑- protocolo, y en consecuencia es en efecto un protocolo de conocimiento cero.

Un protocolo de conocimiento cero es un protocolo en el que dos partes interactúan. La primera de ellas "probador" intenta convencer a la segunda "verificador" que cierta afirmación es cierta y que él/ella conoce el porqué. Después de la interacción la segunda parte queda convencida de la verdad de la afirmación sin que adquiera ningún conocimiento extra más allá de dicha veracidad.

Los ∑- protocolos, son protocolos sencillos en que las dos partes interactúan en tres etapas. Hay métodos bien estudiados y estándars para convertir un ∑- protocolo en un protocolo de conocimiento cero. Los ∑- protocolos tienen la ventaja de que son más fáciles de entender que aquellos y se adaptan bien a diseñar pruebas de autentificación en situaciones prácticas.

Para modelizar el problema en Isabelle hemos usado ideas de Bella (Bella.G) y Paulson, Larry (Paulson.L)

Palabras clave: Isabelle, HOL, conocimiento nulo, Fiat-Shamir, criptografía.

ABSTRACT

The objective of this project titled ”VERIFICACIÓN DE SEGURIDAD DE UN PROTOCOLO CRIPTOGRAFICO UTILIZANDO UN ASISTENTE DE DEMOSTRACIÓN" , is to prove, using Isabelle proof assistant , that the zero knowledge cryptographic protocol called Fiat-Shamir is a ∑- protocol, and therefore a it is indeed a zero knowledge protocol.

A zero-knowledge protocol is a protocol in which two parts interact. The first part “prover” try to convince the second part “verifier” that some claim is true and that he/she knows the reason why. After this interaction the second part get convinced of the truth of the claim, but without getting any extra knowledge or reason why it is true.

∑-protocols can be considered as simple zero-knowledge protocols in which two parts interact in three steps. They are easy to understand and suitable for practical purposes in authentication schemes.

To perform the modelization of Fiat-Shamir protocol in Isabelle we use ideas of Bella, G (Bella.G) and Paulson, Larry (Paulson.L)

Keywords: **Isabelle, HOL, Zero-knowledge, Fiat-Shamir, cryptography.**

Índice

1. Introducción

Este proyecto de fin de carrera tiene el propósito de mostrar a través de herramientas lógicas y matemáticas, la corrección de un protocolo criptográfico de conocimiento nulo establecido entre dos partes. En última instancia podemos decir que ratifica la verdad de una declaración de una de las partes, sin revelar ningún conocimiento adicional a la segunda parte.

En el pasado la construcción, el diseño, y la verificación de seguridad criptográfica de los protocolos se hacıa a mano lo que consumía mucho tiempo y ha llevado en ocasiones a error (véase por ejemplo (Bella.G). Hoy día se tiende a hacer "pruebas formales".

Más concretamente, nos centramos en uno de estos protocolos, el llamado protocolo de Uriel Feige, Amos fiat y Adi Shamir (Feige.U, Fiat.A ,Shamir.A). En lo que sigue, siguiendo la mayoría de la literatura especializada (Trappe-Washington), nos referiremos a este simplemente como protocolo de Fiat-Shamir.

Este protocolo está basado en un problema difícil de la aritmética modular: el hecho de que no se conoce un algoritmo eficiente para calcular raíces cuadras módulo un entero impar compuesto, del que se desconoce su factorización en factores primos.

El protocolo de Feige-Fiat-Shamir se basa en la repetición de tres etapas bien definidas; es uno de los llamados ∑- protocolos.

En última instancia se demuestra que un intruso de la comunicación entre ambas partes de dicho protocolo, no puede extraer el secreto principal de la comunicación.

La herramienta utilizada para la demostración, tiene el nombre de Isabelle, que fue desarrollada por el profesor de lógica computacional de la Universidad de Cambridge, Larry Paulson (Paulson.L), junto con la colaboración de Tobias Nipkow (Nipkow.T) (Universidad Técnica de Múnich) y Makarius Wenzel (Wenzel.M) (Universidad de Paris-Sur).

Esta herramienta nos permitirá por medio de reglas y técnicas elaborar fórmulas matemáticas para ser expresadas en un lenguaje formal y demostrarlas mediante un cálculo lógico. Estas reglas y técnicas se basan en el sistema HOL (Lógica de orden superior), cuya característica más destacable es su alto grado de capacidad de

programación a través del meta-lenguaje ML, que es un lenguaje de programación de propósito general de la familia de los lenguajes funcionales.

El sistema cuenta con una amplia variedad de usos, desde la formalización de las matemáticas puras a la verificación de hardware industrial. En la actualidad muchos sitios industriales y académicos de todo el mundo están utilizando HOL.

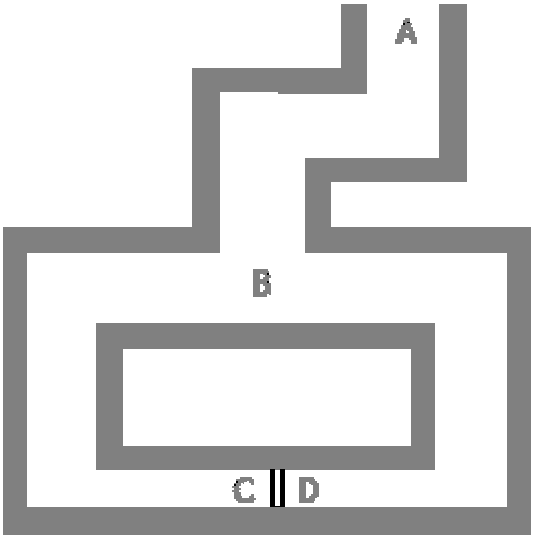
2. Protocolos de conocimiento nulo (zero-knowledge)

En criptografía un protocolo (o prueba) de conocimiento nulo o protocolo (o prueba) de conocimiento cero es un protocolo criptográfico que establece un método interactivo entre dos partes; una de las partes (probador) trata de probar a la otra (verificador) que una declaración (generalmente matemática) es cierta, sin que al final del proceso la segunda parte (verificador) conozca nada más que la veracidad de la declaración, y no sus razones.

Esta idea fue introducida por Goldwasser-Micali-Rackoff en 1989 en " The knowledge Complexity of Interactive Proof Systems", SIAM J. Computing, vol. 18, 1989. Su introducción y desarrollo es una de las razones de que los dos primeros autores hayan sido galardonados en 2012 con el premio "Turing 2012" de la ACM.
(http://www.acm.org/press-room/awards/turing-award-12).

En esta sección explicamos detalladamente en qué consiste una prueba de conocimiento-cero, las características que debe tener, y un esquema muy concreto bajo el que aparecen una gran parte de estas pruebas: los ∑- protocolos. Explicamos estos conceptos centrándonos en nuestro caso de estudio: el llamado protocolo de Fiat-Feige-Shamir.

En el siguiente ejemplo, introducido por Quisquater , Gillou et alt. ("How to Explain Zero-Knowledge Protocols to Your Children". Advances in Cryptology - CRYPTO '89: Proceedings 435: 628–631). Alicia quiere demostrar a Bob que sabe las palabras secretas (clave) para abrir una puerta mágica, sin decir esas palabras delante de Bob (sin enseñarle la clave). Como consecuencia, Bob quedará convencido de que Alicia conoce las palabras secretas, pero no podrá demostrar ante una tercera persona que esto es así. La demostración es la siguiente:



La puerta mágica está en la cueva de Alí-Babá. La cueva tiene una entrada [A] y una bifurcación [B] que llega a ambos lados de la puerta [C y D].

Alicia entra en la cueva, y asegura como demostración que, dado que sabe las palabras mágicas, cuando Bob entre y llegue a la puerta, ella estará en el mismo lugar por el que él entre, dado que puede abrir la puerta y pasar de un lado a otro [C] a [D] antes de que él llegue.

Alicia entra por cualquier camino. Cuando Bob comienza a entrar y llega al punto [B], decide si continuar por la izquierda o por la derecha. Puede gritarle a Alicia por donde va a ir, o sencillamente Alicia oye los pasos de Bob: si escucha que entra por el mismo lado en el que ella está, le espera allí. Pero si entra por el lado contrario, usa las palabras mágicas para abrir la puerta y volverla a cerrar, apareciendo en el lado correcto. Bob llega a la puerta [C] o [D] y encuentra a Alicia, y también comprueba que la puerta está cerrada.

Si este protocolo se realizara sólo una vez, Bob podría pensar que Alicia ha tenido suerte al entrar en la cueva, esperándole en el lado correcto (la probabilidad es del 50 por ciento). De modo que se repite la operación. Alicia vuelve a acertar. Una vez más... y Alicia vuelve a estar allí. Bob puede repetir el protocolo cuantas veces quiera, y tras un número de veces razonable (diez o veinte) le empieza parecer imposible que Alicia haya tenido tanta suerte como para acertar siempre. Si Bob repitiera el protocolo mil veces, la probabilidad de que Alicia acertara por puro azar sería infinitesimal. De modo que es cierto que Alicia sabe algo, aunque no le haya transmitido ese conocimiento a Bob.

Una prueba de conocimiento-cero debe satisfacer las siguientes tres propiedades:

1. **Completitud**: si la declaración es correcta, el "verificador" honesto (esto es, aquel que sigue el protocolo correctamente) quedará convencido del hecho por un "probador" honesto.

2. **Corrección lógica**: si la declaración es falsa, ningún "probador" deshonesto podrá probar al "verificador" honesto que es verdadera. Excepto, con una probabilidad muy baja.

3. **Conocimiento-Cero**: si la declaración es verdadera, ningún "verificador" deshonesto aprende algo más que este hecho. Esto se formaliza mostrando que cada "verificador" deshonesto tiene algún "simulador" que, dado el argumento a probar (y ningún acceso al "probador"), puede producir una copia que "parece ser como" una interacción entre el probador "honesto" y el "verificador" deshonesto.

La investigación en pruebas de conocimiento-cero ha estado motivada por sistemas de autenticación donde una parte quiere probar su identidad a la otra a través de alguna información secreta (por ejemplo un password), pero no quiere que el segundo ente conozca nada de cuál es su secreto.

Demostrar que un protocolo es de conocimiento nulo es difícil. Actualmente se utilizan los ∑-protocolos (estudiados principalmente por Ronald Cramer (Cramer.R), e Ivan Damgard (Damgard.I) que vienen a ser una versión eficiente de los primeros y bien adaptada a las aplicaciones.

2.1 ∑ - Protocolos

Consideramos una relación binaria,
R , que es un subconjunto de {0,1}* x {0,1}*. El conjunto {0,1}* representa el conjunto de cadenas de bits de longitud arbitraria y ***R*** es un conjunto de parejas de enteros no negativos dados por su desarrollo en base 2.

Si ***(v,s) ∈ R*** podemos pensar que ***v*** es un valor concreto de un problema computacional de la aritmética y ***s*** una solución, computacionalmente difícil de calcular.

Imponemos una restricción sobre el tamaño de ***s*** en función de ***v***, para los pares ***(v,s) ∈ R:***

Existe un polinomio T^a (**a** ***∈ N***) , tal que :

Si ***(v,s) ∈ R*** , entonces la longitud(***s***) < longitud(***v***)^a (donde la longitud de un entero es el número de bits de este.)

Los elementos de ***R*** para los que existe ***s*** con ***(v,s) ∈ R***, decimos que forman el lenguaje de ***R***(L_R). Se dice en breve que un ***s***, tal que ***(v,s) ∈ R***, *es testigo de* ***v.*** Como veremos ***s*** no tiene porqué ser el único testigo para ***v*** dado.

Es especialmente interesante el caso de relaciones ***R*** difíciles. Esto es, relaciones en que es fácil construir parejas ***(v,s) ∈ R***, incluso con longitud (número de bits) de ***v*** dado, pero dado un ***v*** tal que existe ***s***, ***(v,s) ∈ R***, no se conocen algoritmos eficientes, ni siquiera probabilistas, que con alta probabilidad construyan un testigo ***s'*** (eventualmente igual a ***s***), tal que ***(v,s') ∈ R***.

Este es el caso de Fiat Shamir como veremos más adelante (véase propiedad. ii en 2.4)

DEFINICIÓN Un protocolo ***(P,V)*** con desafío de longitud ***t*** relativo a una relación ***R***, con input común ***v*** tal que existe ***s*** con ***(v,s) ∈ R***, donde ***s*** es el input privado de ***P***, y ***P*** y ***V*** son algoritmos probabilistas polinomiales , consiste en el siguiente proceso iterativo en tres etapas:

i) ***P*** envía a ***V*** un mensaje ***a***.

ii) ***V*** envía a ***P*** una cadena aleatoria ***e*** de t-bits . (En nuestro caso de estudio será t=1)

iii) ***P*** envía una respuesta ***z*** y ***V*** decide si acepta o rechaza; basado en lo que ***V*** conoce: ***v,a,e,z***.

A la tupla ***(a,e,z)*** se le llama *conversación* del algoritmo ***(P,V)*** con entrada común ***v***.

Un protocolo ***(P,V)*** se dice que es un $\sum$ -protocolo para la relación ***R*** con desafío de longitud ***t***, si se verifican las siguientes propiedades:

1. Completitud : Si ***P*** y ***V*** siguen el protocolo (son honestos) con entrada común ***v***, y entrada privada ***s*** para ***P***, Cuando $(v,s) \in R$, ***V*** siempre acepta.

2. Corrección lógica especial: Para cada entrada común ***v*** y cada par de conversaciones aceptadas con esta entrada ***(a,e,z)*** y ***(a,e',z')*** con $e \neq e'$, (que llamamos colisión)
se puede "de manera eficiente" calcular ***s*** tal que $(v,s) \in R$.

3. Conocimiento cero del verificador honesto: Existe un "simulador" (algoritmo probabilista polinomial), que con entrada ***v*** y valores aleatorios de ***e*** produce una conversación ***(a,e,z)*** que tiene la misma probabilidad que las conversaciones creadas por un honesto ***(P,V)*** (i.e. una pareja ***(P,V)*** que siguiera el protocolo.)

Observaciones:

1. En la tesis de Ronald Cramer (Cramer.R) se pide a la relación ***R*** y al protocolo que verifiquen la siguiente propiedad: si ***v*** es tal que no existe $(v,s) \in R$, entonces no haya colisiones. El caso estudiado de Fiat-Shamir tiene esa propiedad.

2. En general en un $\sum$- protocolo el probador deshonesto (que no conociera ***s***) tiene probabilidad $1/2^t$ de producir una conversación aceptada por el verificador; basta acertar con la cadena aleatoria de bits que ***V*** va a enviar. Nos conformaremos con ver que esto es cierto para ***t=1*** en el protocolo de Fiat Shamir (véase la propiedad iv) en 2.4.

3. Demostraremos que el protocolo de Fiat Shamir es un $\sum$ -protocolo con desafío de longitud 1. (véase Fiat Shamir propiedad iv).

¿Por qué es suficiente trabajar con ∑-protocolos?

Damgard en (Damgard.I) y Ronald Cramer (Cramer.R) en su tesis demuestran que, si la relación R es difícil, a partir de un ∑- protocolo se puede de manera general describir un protocolo de conocimiento cero. Esos argumentos se salen de los límites de esta memoria de Licenciatura y no vamos a incidir en ellos.

En cuanto a la propiedad de conocimiento cero con verificador posiblemente malicioso existen procedimientos para reducirla a una con verificador honesto como por ejemplo en (Goldwasser.S, Micali.S , Rackoff.C)

2.2 Nociones de algebra

2.2.1 Operaciones con congruencias en Z módulo m.

Definición de congruencia

Dado $m \in Z$, $m > 1$, se dice que $a, b \in Z$ son congruentes módulo m si y sólo $si\ m|(a-b)$. Se denota esta relación como $a \equiv b \pmod m$. m es el módulo de la congruencia.

Es importante darse cuenta de que si m divide a $a-b$, esto supone que ambos a y b tienen el mismo resto al ser divididos por el módulo m.

Ejemplos:

$$23 \equiv 2 \bmod 7 \text{ (porque } 23 = 3.7 + 2)$$

$$-6 \equiv 1 \bmod 7 \text{ (porque } -6 = -7.1 + 1)$$

La relación de congruencia como equivalencia. El conjunto de residuos.

La relación de congruencia módulo m es una relación de equivalencia para todo $m \in Z$. Es decir, cumple las propiedades reflexiva, simétrica y transitiva. Como en toda relación de equivalencia, podemos definir el conjunto cociente de las clases de equivalencia originadas por la relación de congruencia. En este caso la relación clasifica a cualquier entero a según el resto obtenido al dividirlo por el módulo m.

Llamaremos Z_m al conjunto cociente de Z respecto de la relación de congruencia módulo m. A la clase de equivalencia de un elemento $a \in Z$ se la denota por $[a]_m$ o simplemente $[a]$.

Para todo $a \in Z$ se tiene que $[a] = [r]$ en Z_m, donde r es el resto de dividir a entre m.Por lo tanto, el conjunto Z_m es finito y tiene m elementos: $Z_m = \{ [0]_m, [1]_m, \dots , [m-1]_m\}$, donde la clase $[i]_m$ representa al conjunto de todos los enteros que son congruentes con $i \bmod m$. A este conjunto cociente se le conoce como el conjunto de restos o residuos (módulo m)

Ejemplo: Siguiendo con el ejemplo anterior, está claro que en Z_7, el número entero *9*, el *16* y el *23* pertenecen todos a la clase $[2]$, y que el entero *-6, el 1 y el 8* pertenecen a la clase $[1]$

Compatibilidad de la relación de congruencia con la suma y el producto

Sean $m \in N$ y $a, b, c, d \in Z$ tales que $a \equiv b \pmod m$ y $c \equiv d \pmod m$. Entonces se cumple que:

$$a + c \equiv b + d \pmod m$$

$$a \cdot c \equiv b \cdot d \pmod m$$

Consecuentemente, el resto de la suma es congruente con la suma de restos, y el resto del producto es congruente con el producto de restos. Además podremos sumar y multiplicar clases de equivalencia (residuos) porque es indiferente el representante que se elija de cada clase a la hora de operar: el resultante de la operación siempre será un representante de la misma clase resultado.

Lo dicho anteriormente muestra que la adición “natural” y la “multiplicación natural” en el conjunto cociente para la relación de equivalencia de congruencia módulo m que notaremos Z_m, dan a éste estructura de anillo.

Por último, a partir de ahora utilizamos la siguiente notación, $a \bmod m$ significará el único entero $j \in \{0,..,n-1\}$, tal que $a \equiv j \pmod m$.

Obviamente se verifican las siguientes igualdades en Z^+:

$$((a \bmod m) + (b \bmod m)) \bmod m = (a + b) \bmod m$$
$$((a \bmod m) \times (b \bmod m)) \bmod m = (a \times b) \bmod m$$

Recordemos que la identidad de Bezout nos permite afirmar que si $a,n \in Z$ y $mcd(a,n)=1$, existe $a' \in Z$ tal que $a.a' \bmod n=1$. Tal entero menor que n, $a' \bmod n$, es único con esa propiedad y lo llamaremos ***inverso de a módulo n***. Escribiremos también $a^{-1} \bmod n: = a' \bmod n$.

Por último; el subconjunto de Z_n formado por las clases de restos módulo n cuyo representante entre $0,...,n-1$ es primo con n, y por consiguiente tiene inverso módulo n se denotará Z_n^*

El principio de inducción matemática

En las demostraciones usando Isabelle, a menudo tenemos la necesidad de utilizar el principio de inducción matemática, que dice lo siguiente:

Dado un enunciado P dependiente de un parámetro $n \in Z$, supongamos que se demuestra que:

i. $P(n_0)$ es cierto para un cierto $n_0 \in Z$
ii. Siempre que $P(k)$ es cierto para cualquier entero $k \geq n_0$, entonces es cierto para el siguiente entero $P(k+1)$.

Entonces podemos afirmar que $P(n)$ es cierto $\forall n \in Z$ con $n \geq n_0$

La demostración de un enunciado matemático mediante el principio de inducción, consiste primero en probar que para el caso básico inicial de n_0 se satisface el enunciado. Después supondremos que se cumple para un determinado valor k mayor que n_0 (a esta suposición se la llama ***hipótesis de inducción***) y comprobamos si se satisface también para **k+1**. Si también se cumple el enunciado para **k+1**, entonces quedará demostrado que se cumple el enunciado para todo entero mayor o igual que n_0.

El principio de inducción fuerte

A veces resulta conveniente tomar como hipótesis de inducción la suposición de que el resultado es cierto para **todos** los valores anteriores relevantes $n \leq k$, en lugar de suponer cierto sólo el caso $n = k$. Esta variante del principio de inducción se la suele llamar ***principio de inducción fuerte***, que se formularía así:

Dado un enunciado $P(n)$ dependiente de un parámetro $n \in Z$, supongamos que se demuestra que:

i. **$P(n_0)$** es cierto para un cierto $n_0 \in Z$
ii. Siempre que $P(m)$ es cierto para cualquier entero $n_0 \leq m \leq k$, entonces $P(k+1)$ es cierto.

Entonces podemos afirmar que $P(n)$ es cierto $\forall n \in Z$ con $n \geq n_0$

2.2.2 Teorema chino de los restos

La forma original del teorema, contenida en un libro del siglo III por el matemático chino Sun Tzu y posteriormente publicado en 1247 por Qin Jiushao, es un enunciado sobre congruencias simultáneas.

Supongamos que $n_1, n_2, \ldots, n_k$ son enteros coprimos dos a dos. Entonces, para enteros dados $a_1, a_2, \ldots, a_k$, existe un entero x que resuelve el sistema de congruencias simultáneas.

$$x \equiv a_1 \pmod{n_1}$$

$$x \equiv a_2 \pmod{n_2}$$

.....

$$x \equiv a_k \pmod{n_k}$$

Más aún, todas las soluciones x de este sistema son congruentes módulo el producto $N=n_1 n_2 \ldots n_k$

Observemos que, para cualquier solución x del sistema:

$$mcd(a_i; n_i) = 1;\ i = 1..r\ ,\ mcd(x,N) = 1$$

Una consecuencia del Teorema Chino de los restos que utilizaremos más adelante, en un caso particular (el caso de dos factores primos) es la que mencionamos a continuación:

DEFINICIÓN Dado $n \in Z^+$,y un entero $a \in Z$ decimos que $a \bmod n$ tiene raíz cuadrada módulo n, si existe $c \in Z$ tal que $a \equiv c^2 \pmod{n}$. Decimos entonces que $c \bmod n$ es una *raíz cuadrada módulo n* de **a mod n .**

PROPOSICIÓN Sea $n \in Z^*$, $n = \prod_{i=1}^{r} p_i$ donde los p_i son primos distintos. Para cada entero $a \in Z$, tal que ***mcd(a, n) = 1***, ***a mod n*** tiene raíz cuadrada módulo ***n*** si y sólo si para todo ***i = 1,.. r, a mod*** p_i tiene raíz cuadrada módulo p_i y en tal caso, ***a mod n*** tiene exactamente 2^r raíces cuadradas módulo n. En particular si se conoce la factorización de ***n*** en producto de primos distintos, y se conoce una raíz cuadrada de ***a mod*** p_i para cada ***i = 1..r***, se calculan fácilmente las 2^r raíces cuadradas módulo n de ***a mod n***.

Como ejemplo de esta afirmación veremos un ejemplo.

Ejemplo: *calcular raíces cuadradas mod n conociendo mod cada factor primo y conociendo la factorización de n.*

Por ejemplo ***n=7 x 11.*** Calculamos las raíces de ***1 mod n*** y calculamos los ***c (mod n)*** tales que $c^2 \equiv 1 \pmod 7$ y $c^2 \equiv 1 \pmod 1$. Las soluciones módulo 7 son***: c ≡ 1(mod 7) , c ≡ -1=6(mod 7);*** y módulo 11 son ***: c ≡ 1 (mod 11), c ≡ -1=10 (mod 11).***

Ahora combinamos dos a dos de cuatro formas posibles las congruencias: Así buscamos encontrar ***x*** tal que ***x ≡ 1 (mod 7), x ≡ 1 (mod 11).*** Utililizando la demostración del teorema que se basa en la identidad de Bezout: ***-3x7+2x11 = 1*** cuya solución es ***x ≡ 1(mod 77)*** (que sale de 2x11x1-3x7x1=1).

Buscamos ahora un ***y*** tal que: ***y ≡ 6 (mod 7), y ≡ 10(mod 11)*** cuya solución es ***y ≡ 76 (mod 77)*** (que sale de 2x11x6-3x7x10 ≡-78, -78=-1(mod 77) ≡ 76 (mod 77)).

Buscamos ahora un ***z*** tal que: ***z ≡ 1 (mod 7), z ≡ 10 (mod 11)*** cuya solución es ***z ≡ 43 (mod 77)*** (que sale de 2x11x1-3x7x10 =-188, ***-188 (mod 77) ≡ 43.***

Por último buscamos ***u*** tal que ***u ≡ 6 mod 7; u ≡ 1 mod 11*** , cuya solución es ***u ≡ 34 (mod 77)*** (que sale de 2x11x6-3x7x1=111, ***111 ≡34 (mod 77)***).

Así las raíces de *1 (mod 77)*, aparte de ***1, -1 mod 77*** son ***43 y 34*** (43 ≡ -34 mod 77).

2.3 Complejidad de los algoritmos

La seguridad de la Criptografía de Clave Pública se basa de manera esencial en la existencia de problemas NP de la Aritmética. Por ello introducimos aquí la complejidad binaria de algoritmos sobre los enteros y en el punto siguiente discutiremos la dificultad del cálculo de raíces cuadradas modulares.

En términos generales la complejidad de algoritmos en algebra se mide en función de ciertos parámetros de las entradas llamados parámetros de la complejidad. Tales parámetros, son por ejemplo, el número(s) de bits de la(s) entradas si se trata de algoritmos sobre los números enteros, o el grado o grados o/y número de variables si se trata de algoritmos sobre polinomios.

La complejidad de un algoritmo será el número de pasos que da el algoritmo desde que recibe una entrada hasta que termina produciendo una salida, en función del tamaño de aquella, es decir; del tamaño de, o de los, parámetros de la complejidad de la entrada. En aritmética sobre los enteros usualmente el parámetro de la complejidad es el máximo número de bits de las entradas. Cada paso simple o paso de coste 1 es una operación en bits. Sin embargo cuando trabajamos con cuerpos y polinomios, interesa tomar como parámetros de la complejidad el grado de los polinomios y el número de variables, considerando así paso de coste 1 cada operación aritmética en el cuerpo.

Al estimar la complejidad de un algoritmo de la aritmética de enteros con frecuencia nos interesará únicamente una medida asintótica del peor caso, esto es, cuando los valores de los parámetros son suficientemente grandes, lo que usaremos como sigue.

> ***Dadas $f, g : N \to N$ decimos que g es una O-grande de f ; $g = O(f)$, si $\exists N, C \in N$, s.t. $\forall n \geq N, f(n) \leq C\, g(n)$.***

Un entero ***n*** lo representamos en base B, y casi siempre en base 2. Llamamos B-bits y respectivamente bits sus dígitos en dichas bases Si tal entero es la entrada de un algoritmo, su parámetro de la complejidad como hemos dicho, será el número de B- bits, o el número de bits. Nótese que $\log_B n \times \log_2 B = \log_2 n$[1]; así para una medida asintótica de la complejidad podemos tomar como parámetro de la complejidad indistintamente $\log_2 n$ o $\log_B n$, ya que $\log_2 B$ es una constante. Además si ***n*** es un k-bit : $2^{k-1} \leq n < 2^k$ y $[\log n] + 1 = k$. por consiguiente también podemos tomar log n como parámetro de la complejidad para una medida asintótica .

Un algoritmo de la aritmética de enteros es eficiente si su complejidad binaria está acotada asintóticamente por un polinomio en el número de bits de las entradas . Esto es su complejidad binaria es del tipo $O(\log n)^e$, con $e \in N$, siendo n la mayor de las entradas. Para que sea útil en Criptografía el exponente e debe ser un número natural pequeño, no más de 5, por ejemplo.

2.3.1 Raíz cuadrada, raíz cuadrada modular y residuos cuadráticos

Comencemos con un algoritmo sencillo y eficiente para calcular una aproximación a la raíz cuadrada de un entero no negativo (esto es, calcular la parte entera de su raíz cuadrada).

Dado $n \in Z^+$, calcular $\sqrt{n}$ tiene complejidad binaria $O(\log^3 n)$. Averiguar si n es una potencia pura, y escribirlo como tal cuesta $O(\log^4 n)$.

DEMOSTRACIÓN Sea $n: 2^{l-1} \leq n < 2^l$, se calcular en función de l el número de bits de la parte entera de la raíz cuadrada de n. En efecto $si\ l= 2k$,

$$2^{(l-1)/2}= 2^{k-1/2} \leq \sqrt{n} < 2^{l/2} = 2^k$$

y asi, $2^{k-1/} \leq [\sqrt{n}] < 2^k$. Por tanto el número de bits de $[\sqrt{n}]$ es $k = l/2$. Análogamente se prueba que si l es $2k + 1$, el número de bits de $[\sqrt{n}]$ es $(l+1)/2$.

Llamemos q a este número de bits. Se considera el entero a de q bits que tiene como segundo bit significativo el 1 y es el menor con esta propiedad. Es decir $a := 1100…000$, se calcula a^2. Si es $n < a^2$, entonces el segundo bit más significativo de $[\sqrt{n}]$ es *0*. Si por el contrario $a^2 \geq n$, el segundo bit más significativo de $[\sqrt{n}]$ es *1*. Se continúa determinando el tercer bit más significativo, etc...En cada paso hacemos una multiplicación y una resta de l- ***bits***, y esto lo hacemos q veces (el número de bits de la parte entera de la raíz buscada), es decir aproximadamente $l/2$ veces. En resumen la complejidad está acotada por $O(\log^3 n)$.

Para un r fijado, calcular la raíz r-esima, o la parte entera de ésta funciona de manera análoga, y con la misma cota asintótica que el cálculo de $[\sqrt{n}]$.

Ahora supongamos que queremos averiguar si n es potencia pura, es decir $si\ n=k^r$ y calcular r y k. Es decir intentamos calcular si n tiene raíz r-esima exacta para algún r. Si la respuesta es afirmativa, debe *ser* $r \prec \log_2 n$. Así, el coste total será, $O(\log^3 n)$.

Residuos cuadráticos.

El problema de calcular una raíz cuadrada de un entero n módulo un numero primo p, o averiguar si ésta existe, resulta ser un problema clásico y difícil si no se imponen condiciones adicionales sobre el numero primo ***p***.

Gauss para desarrollar su estudio de residuos cuadráticos utilizo la idea de Legendre de asociar a un entero ***n*** mod ***p***, donde ***p*** es un numero primo, o un valor *0*, *1* ó *−1*, que denotaremos $\left(\frac{n}{p}\right)$ y llamaremos símbolo de Legendre de ***n*** con respecto a ***p***, según se tenga respectivamente que: ***n*** es múltiplo de ***p***, que ***n*** es un cuadrado mod ***p*** o que ***n*** no es un cuadrado mod ***p***.

Resulta que el símbolo de Legendre es computable de modo eficiente, como mostramos a continuación.

PROPOSICIÓN Dado $n \in Z^+$, y ***p*** número primo impar, se verifica $\left(\frac{n}{p}\right) = n^{(p-1)/2} \mod p$.

DEMOSTRACIÓN . Si $n = 0 \mod p$ ambos términos son *cero módulo* ***p***.

Supongamos que $n \mod p \in Z_{p^*}$, y sea $g \in Z$, tal que $g \mod p$ genera el grupo cíclico $\in Z_{p^*}$.Entonces $n \equiv g^r \mod p$. Si ***r*** es par, ***n*** es un cuadrado. Recíprocamente si ***n*** es un cuadrado $\mod p$, $n \equiv c^2 \mod p \equiv g^{2s} \mod p$. Así, en el grupo Z_{p^*} tenemos $g^r \equiv g^{2s} \mod p$, y como el orden de **g** es $p - 1$, tenemos $r \equiv 2s \mod p - 1$. Como $p - 1$ es par ***r*** es par. De aquí se deduce el hecho de que en Z_{p^*} hay exactamente tantos cuadrados como no cuadrados a saber $(p - 1)/2$. Sea $n \equiv g^{\alpha} \mod p$. Entonces $g^{\alpha(p-1)/2} \equiv 1 \mod p$, si y solo si $p - 1$, que es el orden de $g \mod p$, divide a $\alpha(p - 1)/2$ y por tanto si y solo si α es par, si y solo si $n \mod p$ es un cuadrado.

De este modo el averiguar si $n \mod p$ tiene raíz cuadrada se puede resolver con complejidad igual a la de calcular una exponenciación *módulo* ***p***, que con el algoritmo de exponenciación rápida (también llamado de agrupamiento de cuadrados) cuesta $O(\log^3 p)$. Otra cosa distinta es, en caso de que tenga raíz cuadrada calcularla.

Si $p \equiv 3 \mod 4$ y ***n*** es un cuadrado módulo ***p***, se verifica que $n^{(p-1)/2} \equiv 1 \mod p$. Llamando $c := n^{(p+1)/4}$ tenemos que $c^2 \equiv n^{(p+1)/2} \equiv nn^{(p-1)/2} \equiv n \mod p$. Por tanto $\pm c \mod p$ son las raíces de n mod p. No hay más porque en el cuerpo Z_{p^*}, el polinomio $x^2 - n$ tiene como mucho dos raíces.

Si $p \equiv 1 \mod 4$, y se verifica $n^{(p-1)/2} \equiv 1 \mod p$, no es fácil calcular dichas raíces. Hay algoritmos probabilıstas eficientes pero no deterministas. El algoritmo de Shanks, reduce el problema de calcular raíces cuadradas *módulo* ***p***, a calcular un ”no residuo cuadrático“ *módulo* ***p***. Si $p = 3 \mod 4$, entonces nos vale el *-1*.

En general, en el caso $p \equiv 1 \mod 4$ no hay un algoritmo determinista para calcular un n tal que $\left(\frac{n}{p}\right) = -1$. Un algoritmo probabilista consiste en elegir n ≺ p aleatoriamente y utilizar reiteradamente la última proposición cambiando el n si el miembro de la derecha sale un 1. El algoritmo tiene probabilidad *1/2* de acertar con un no residuo cuadrático cada vez.

Aunque hay tantos residuos cuadráticos módulo p como no residuos cuadráticos su distribución es ”caótica“ en el sentido de que su aleatoriedad no está clara. En este desorden se basan muchos sistemas criptográficos. Por ejemplo se puede demostrar que para cada $M \in Z^+$ existe un primo p tal que los números $1, 2, \ldots, M$ son todos residuos cuadráticos módulo p.

Por último tratemos ahora de convencernos que calcular raíces cuadradas módulo un entero impar producto de dos primos es tan difícil como factorizar dicho entero. Este último problema es un problema para el que no se conoce hoy por hoy algoritmo polinomial determinista (ni siquiera polinomial probabilista)

Concretamente sea $n \in Z^+$ que es producto de dos primos impares distintos $n = pq$ y supongamos que tenemos un algoritmo eficiente que dado un $a \mod n \in Z_{n^*}$ con raíz cuadrada en Z_{n^*} nos devuelve aleatoriamente una de las cuatro raíces cuadradas de $a \mod n$, entonces tenemos un algoritmo (probabilista) eficiente para factorizar n. Es decir calcular raíces cuadradas mod n es ”probabilistamente equivalente“ a factorizar n. Como este último problema se considera muy difícil, este es igualmente difícil y en eso se basan algunos sistemas criptográficos de clave pública.

En efecto, imaginemos que tenemos una máquina a la que damos como entrada valores $x_c := c^2 \mod n$ para $0 < c < n$ aleatorios con $mcd(c, n) = 1$ y nos devuelve unas veces c otra $-c$ y otras d ó $-d$, siendo estas dos últimas las dos raíces de x_c en Z_{n^*} ; cada uno de los cuatro caso con probabilidad *1/4*. Cada vez que nos devuelve una raíz y calculamos $mcd(y + c, n)$ y $mcd(y - c, n)$. Si nos devuelve $y = c \mod n$ el cálculo anterior da respectivamente 1(ya que $mcd(c,n) = 1$) y n (ya que $mcd(0,n) = n$).Si nos devuelve $y = -c \mod n$ el mismo calculo da *n* y *1* respectivamente. Pero si y es distinto de $\pm c \mod n$, lo cual sucederá con probabilidad *1/2*, como $y^2 \equiv c^2 \mod n$, se tiene que $mcd(y + c, n)$ y $mcd(y - c, n)$ son divisores propios de n, es decir son p y q ó q y p. En este caso hemos factorizado n.

2.4 El protocolo Fiat-Shamir

El protocolo Fiat-Shamir es uno de los protocolos de Conocimiento-cero más importante. La seguridad de este algoritmo se basa en el hecho de que en la práctica es casi imposible calcular raíces cuadradas en Z_{n*} sin conocer la factorización de n.

Utilizaremos enteros que son producto de dos números primos distintos y grandes a los que llamamos enteros RSA. Decimos que tal entero es de "Blume" si los dos primos son congruentes con 3 módulo 4. Por lo que hemos visto antes, en tal caso, conocida la factorización de ***n*** calcular las cuatro raíces de un número mod n que de hecho las tenga "es sencillo " basta calcularlas módulo los primos y utilizar el Teorema Chino de los Restos. Si no se conoce dicha factorización será un problema computacionalmente difícil.

Para aplicar la formalización de los ∑-protocolos, consideremos la siguiente relación binaria asociada a Fiat-Shamir

$$FS=\{ (v,s) \in N , 1 = s^2.v \bmod n \}$$

Nótese que si $(v,s) \in FS$, es $mcd(v,n)=1$ y $mcd(s,n)=1$, debido a que $s.v = 1 \bmod n$. Entonces existe v' tal que $v.v' \bmod n=1$, y análogamente existe s' tal que $s.s' \bmod n=1$. Llamando $v^{-1} \bmod n := v' \bmod n$, y llamando $s^{-1} \bmod n= s' \bmod n$, y de forma natural $s^{-2} \bmod n:= s^{-2} \bmod n$; podemos escribir:

$$FS= \{ (v,s) \in N , \quad mcd (v,n)=1, \ s^2 \bmod n= v^{-1} \bmod n \}$$

o bien

$$FS=\{ (v,s) \in N , \quad mcd (s,n)=1, \ s^{-2} \bmod n= v \bmod n \}$$

El protocolo de Fiat-Shamir consta (como la mayoría de los protocolos criptográficos) de dos fases, la fase de generación de clave y la fase de aplicación. Participan dos partes P (probador) y V (verificador)

En la fase de generación P genera 2 números primos grandes **p** y **q** y forma el producto **n = pq**.
El número n es público, mientras que **p** y **q** solo los puede conocer P. Después P elige un número **s** tal que $mcd(s,n)=1$, y construye $\mathbf{v := s^{-2} \bmod n}$.

El número **s** es el secreto individual de P, mientras que con la ayuda de **v** (la marca de identificación) se puede verificar si una persona conoce o no el secreto. Esto significa en particular que **s** debe permanecer secreto, mientras que **v** debe ser público.

En la fase de aplicación P debe convencer a V de que conoce el secreto **s**. Para ello P y V siguen el siguiente protocolo:

- P elige aleatoriamente un elemento **r** de $\mathbf{Z_n}^*$ y lo eleva al cuadrado módulo n: **x := r^2 mod n**. Después P manda a V el valor **x**.
- V elige aleatoriamente un bit **b** y lo manda a P.
- Si **b = 0** P manda el valor **y := r** a V
- Si **b = 1** P manda el valor **y := rs mod n** a V

- V verifica esta respuesta. En caso de que **b = 0** comprueba que **y^2 mod n = x**.

En el caso **b = 1** prueba el si la igualdad **y^2 v mod n = x**

OBSERVACIONES Y PROPIEDADES:

i) Veamos que nuestra relación admite distintos testigos. Es decir dado ***n*** un entero que es producto de dos primos ***n=p.q*** impares. Y dado ***(v,s) ∈ FS***, también ***(v, n-s) ∈ FS*** y existen dos raíces más de ***v mod n; s', n-s'*** (que no se saben calcular eficientemente ni siquiera conociendo ***s***). Así también ***(v,s'), (v, n-s') ∈ FS.***

ii) La relación FS es difícil. Ya hemos comentado la dificultad de calcular raíces cuadradas de ***v mod n*** (o de **v^{-1} mod n,** sabiendo que las tiene) .

iii) En primer lugar nótese que una falsa P' (que no conoce s) si acierta el valor de ***b*** puede engañar a V. En efecto, si acierta que va a ser ***b = 0***, envía en la primera etapa (para r aleatorio) ***$x= r^2$ mod n***, en la segunda ***y=r*** . Finalmente V verifica ***y^2 mod n = x.*** Por el contrario si acierta que será ***b = 1***, envía primero ***$x= r^2$ v mod n***, y luego ***y=r mod n.*** Finalmente V verifica ***y^2 v mod n =x mod n***. Se puede entonces engañar en una ronda con la probabilidad de ½

iv) El anterior es un ∑-protocolo con desafío de longitud 1 para la relación FS. Esto lo probamos a continuación.

El protocolo de Fiat-Sahmir consiste en repetir un número suficiente alto de veces este ∑-protocolo.

Completitud: Cuando P es honesto (conoce el secreto s y sigue el protocolo) V acepta la conversación. Esto es obvio.

$$y^2 v^b \equiv (rs^b)^2 v^b \equiv r^2 (s^2 v)^b \equiv r^2 \bmod n$$

Corrección lógica especial:

Si ***(x,b,y) (x,b',y'***) (con ***b*** y ***b'*** distintas) son conversaciones admitidas por V; quiere decir que $y^2 \bmod n = xv^b \bmod n$, $y'^2 \bmod n = xv^{b'} \bmod n$, entonces ***si b=1, b'=0*** (o viceversa) dividimos $xv^b \bmod n$ entre $xv^{b'} \bmod n$ en Z_{n*} y tenemos ***v.***Tenemos ***v*** expresado como un cuadrado es decir obtenemos $v = (y\, y'^{-1})^2 \bmod n$ Llamando $s' := (y\, y'^{-1}) \bmod n$, tenemos que $s'^2 v \equiv 1 \bmod n$. Así, P conoce una raíz cuadrada de v^{-1}, a saber ***s'***. Es fácil ver que P conseguirá que V verifique positivamente todas sus conversaciones, es decir las acepte, utilizando la ***s'*** como debiera de utilizar la ***s*** en el protocolo.

Conocimiento-Cero de verificador honesto: Un simulador M puede realizar un dialogo con V de la siguiente manera:

- M elige aleatoriamente un bit **c** y un número **r**; después calcula **x := r^2 v^c mod n** y envía **x** a V
- V responde con un bit **b**
- si **b = c**, entonces M manda el mensaje **y = r** a V. En este caso si la verificación tiene éxito se tendrá:

$$x \equiv r^2 v^c \equiv r^2 v^b \equiv y^2 v^b \pmod n$$

La terna (***x; b; y***) representa un paso de esta simulación de la conversación.

- Si **b** es distinto de **c** entonces todos los mensajes enviados serán borrados y la simulación en esta ronda debe empezar de nuevo.

Tanto en el diálogo original, como en el previo aparecen el mismo número (aleatorios) de ternas ***(x; b; y)*** para las cuales la igualdad $\mathbf{xv^b = y^2}$ es válida. Ambos diálogos no pueden ser diferenciados por un observador.

El protocolo de Fiat-Shamir es especialmente útil para protocolos de autentificación.

3. El asistente de demostración ISABELLE/HOL

3.1 HOL (Higher Order Logic)

HOL (Higher Order Logic) se ha utilizado para la demostración de teoremas definidos en Isabelle. Con este fin, la lógica formal se interconecta a un lenguaje de programación de fines generales, ML (META-LENGUAJE). HOL fue pensado inicialmente para llevar a cabo la especificación y la verificación de los diseños hardware. Sin embargo, la lógica no restringe su uso al hardware; HOL se ha aplicado a muchas otras áreas.

3.2 ¿Qué es Isabelle?

Isabelle es un conocido demostrador de teoremas, desarrollado en la Universidad de Cambridge (Larry Paulson) y en la Universidad Técnica de Múnich (Tobías Nipkow).

Isabelle es un demostrador genérico de teoremas escritos en ML estándar. Es genérico en el sentido que los usuarios pueden definir su propia lógica.

El uso de Isabelle consiste en definir fórmulas matemáticas en un lenguaje formal, para luego razonar sobre ellas, utilizando métodos de deducción de la lógica. Las áreas de aplicación más importantes son, la formalización de demostraciones matemáticas, la verificación formal y la demostración de propiedades, es decir, la corrección de circuitos hardware y de programas software y la demostración de propiedades de lenguajes, programas y protocolos.

Comparado con herramientas similares, la característica que distingue a Isabelle es su flexibilidad. La mayoría de los programas demostradores de teoremas estan construidos basados en un sólo cálculo formal. Isabelle es capaz de aceptar una gran variedad de cálculos formales. La versión que se presenta se basa en HOL pero también lo hace en la teoría axiomática y en varios formalismos más.

3.3 ¿Qué ofrece Isabelle?

Isabelle proporciona una ayuda excelente para demostrar teoremas, se pueden introducir nuevas notaciones, usando símbolos matemáticos normales. Las demostraciones se pueden escribir en una notación estructurada basada en el estilo tradicional de la demostración (no varía mucho de una demostración en un documento), o directamente mediante comandos. Las definiciones y las pruebas pueden incluir la fuente de LaTeX (LaTeX es un sistema de composición tipográfica de textos de alta calidad), de la cual Isabelle puede generar automáticamente un documento.

La limitación principal de estos sistemas de demostraciones, es que los teoremas requieren mucho esfuerzo y tener cierta habilidad en ello. Isabelle incorpora algunas herramientas para mejorar la productividad del usuario automatizando algunas partes del proceso de la demostración. Con Isabelle se pueden realizar demostraciones de fórmulas matemáticas.

Isabelle viene con una gran biblioteca, repleta de teorías de las matemáticas formalmente verificadas, incluyendo la teoría elemental de números (por ejemplo, la ley del gauss de la reciprocidad cuadrática), de análisis (características básicas de límites, de derivadas y de integrales), de álgebra.

Isabelle ofrece un lenguaje preciso a la hora de mostrar los resultados de las demostraciones de forma que permite tanto a los usuarios como a los ordenadores comprender los resultados de dichas demostraciones. Isabelle usa la interfaz de Jedit, que facilita la tarea de escribir las definiciones de las demostraciones.

3.4 Interactuando con ISABELLE

En esta sección, explicaremos y detallaremos la estructura y conceptos de Isabelle/HOL relevantes para el proyecto.

3.4.1 Estructura de una Teoría

Trabajar con Isabelle implica crear teorías. En líneas generales, una teoría es una colección con nombre, de tipos, funciones, y teoremas, de la misma manera que un módulo en un lenguaje de programación o una especificación en un lenguaje de especificación. El formato general de una teoría es la siguiente:

```
theory T
imports B1 ... Bn
begin
declarations, definitions, and proofs
end
```

B1...Bn son los nombres de teorías existentes en Isabelle y que se utilizan en ***T***, la cual está compuesta por las declaraciones, las definiciones, y las demostraciones que representan los conceptos nuevamente introducidos (tipos, funciones etc.) y las demostraciones sobre ellas. Para evitar conflictos (nombres de identificadores iguales en varias Teorías), los identificadores se pueden escribir mediante nombres cualificados por la teoría como en ***T.id*** y ***B.id***. Cada teoría ***T*** debe residir en un archivo nombrado ***T.thy***.

La colección de teorías de HOL está disponible online
http://isabelle.in.tum.de/library/HOL/

3.4.2 Tipos, términos y fórmulas

En la redacción de una teoría se definen los tipos, los términos y fórmulas de **HOL**. **HOL** es una lógica con tipos que incluye un lenguaje de programación que se asemeja a lenguajes de programación funcionales como **ML** o ***Haskell***.

Tipos

En HOL podemos encontrar los siguientes tipos de datos:

- Tipos básicos:
  En particular, el tipo booleano ***bool***, con los valores ***True*** y ***False***, y el tipo ***nat***, de los números naturales.

- Constructores de tipos:
  En particular, el tipo ***list***, que define el tipo listas, y el tipo ***set***, que define el tipo de conjuntos. Los constructores de tipo tienen notación posfija, es decir, ***nat list*** define el tipo de lista cuyos elementos son números naturales.

- Tipos de funciones:
  En HOL, únicamente podemos encontrar funciones totales. Las funciones se denotan con "***=>***". Como es habitual en los lenguajes funcionales, una función de tipo ***t1 => t2 => t3*** significa ***t1 => (t2 => t3)***

- Variables de tipo:
  Se denotan como ***'a***, ***'b*** etc. Estas variables dan lugar a tipos de funciones polimórficas como ***'a => 'a***, que representa el tipo de la función identidad.

Términos

Los términos están constituidos como en la programación funcional, aplicando las funciones a sus argumentos. Si ***f*** es una función de tipo ***(t1 => t2)***, y ***t*** es un término de tipo ***t1*** entonces ***f t*** es un término de tipo ***t2***. HOL también soporta funciones de operadores infijos como ***+*** y algunos conceptos básicos de la programación funcional, como las expresiones condicionales:

If b then t1 else t2

Aquí ***b*** es de tipo booleano, ***t1*** y ***t2*** son del mismo tipo.

Formulas

Las fórmulas son términos de tipo booleano. Hay constantes básicas ***True*** y ***False*** y las conectivas lógicas usuales (en orden decreciente de prioridad): **¬, ^, ˅** y **→**, todas las cuales (excepto la unaria ¬) asocian a la derecha.

Un problema especial para principiantes puede ser la prioridad de operadores. Hay que usar paréntesis adicionales. Por ejemplo, ***A → B → C*** significa ***A → (B → C)***. Por otro lado, ***A ^ B →C*** significa ***(A ^ B) →C***.

Los cuantificadores son definidos como ***(∀x. P)*** para el cuantificador universal y ***(∃ x. P)*** para el existencial. También existe la notación ***(∃ !x. P)*** que nos dice que únicamente existe solo un elemento que cumple la propiedad ***P***. Los cuantificadores anidados pueden ser abreviados: ***(∀x y z. P)*** significa ***(∀x. ∀y. ∀z. P)***

3.4.3 Variables

Isabelle distingue variables libres y ligadas, como es habitual. Las variables ligadas se renombran automáticamente para evitar conflictos de nombres con las variables libres.

Además, Isabelle tiene un tercer tipo de variable, llamada variable esquemática, la cual debe tener una '***?***' como su primer carácter. Desde el punto de visto lógico, una variable desconocida es una variable libre. Funciona como un argumento que puede ser sustituido por un término cuando se usa en el teorema.
La mayoría de las veces se puede y se debe ignorar las variables desconocidas y trabajar con las variables normales. Aunque, no debe de sorprender que después de haber acabado la demostración de un teorema, Isabelle convertirá sus variables libres en variables desconocidas. Esto indica que Isabelle sustituirá automáticamente las variables desconocidas adecuadamente cuando se utilizan el teorema de alguna otra prueba. Hay que tener en cuenta que para facilitar la lectura que a menudo se quita la '***?***' al mostrar un teorema.

3.4.4 Ejemplo de construcción de una Teoría

En este apartado explicaremos paso a paso el detalle del contenido de la siguiente teoría.

```
theory ToyList
imports Datatype
begin

datatype 'a list = Nil                          ("[]")
                 | Cons 'a "'a list"            (infixr "#" 65)

(* This is the append function: *)
primrec app :: "'a list => 'a list => 'a list"  (infixr "@" 65)
where
"[] @ ys       = ys" |
"(x # xs) @ ys = x # (xs @ ys)"

primrec rev :: "'a list => 'a list" where
"rev []        = []" |
"rev (x # xs)  = (rev xs) @ (x # [])"
```

Ilustración 1: Ejemplo de construcción de una teoría

ToyList es el nombre de la Teoría, que utiliza otra teoría llamada ***Datatype*** mediante el comando ***imports***.

A continuación se define el tipo de datos ***list***, que definirá una lista de elementos. Para construir la lista se definen dos constructores de tipo ***Nil*** y ***Cons***. ***Nil*** crea la lista vacia ***[]***, y ***Cons*** añade un elemento a una lista dada. Por ejemplo, el termino ***Cons*** **True** (***Cons*** **False** ***Nil***) es un valor de tipo "***bool list***", es decir, una lista de elementos booleanos.

Como esta notación suele ser difícil de manejar, la declaración del tipo de datos introduce una sintaxis alternativa. En vez de ***Nil*** y ***Cons x xs***, se utilizará ***([])*** y ***(x # xs)*** respectivamente. La notación ***infixr*** significa que el operador ***"#"*** es infijo y asocia la derecha. Mientras que el número 65 determina el orden de prioridad del operador.

Por último, dos funciones, ***app*** y ***rev*** son definidas recursivamente. En Isabelle, el orden de definición es importante ya que si una función necesita utilizar otra, esta debe de estar definida previamente.

Cada definición de una función es de la forma:

primrec nombreDeLaFuncion **::** TipoDeLaFuncion **(sintaxis opcional)** **where** ecuaciones

Las ecuaciones de cada función deben estar separadas mediante el carácter '**|**'.

3.4.5 Ejemplo de construcción de un Teorema/Lema

La construcción de un teorema/lema tiene la siguiente estructura:

theorem/lemma theorem/lemma's name ***[***optional rules***]: "***theorem/lemma body**"*

La palabra ***theorem/lemma,*** indica a Isabelle que se va a definir un teorema o un lema. Esta palabra es seguida del nombre del teorema/lema en cuestión. Este nombre servirá para futuras referencias que se podrán utilizar para la demostración de otros teoremas/lemas.

Después del nombre del teorema/lema, podemos indicar a Isabelle que lo incluya en algunas reglas que son utilizadas en las demostraciones, para que cuando se apliquen dichas reglas, se pueda utilizar este teorema/lema en dicha demostración. Estas reglas deben de ser incluidas entre corchetes.

Finalmente, se incluye entre comillas, el cuerpo del teorema/lema en notación de isabelle.

A continuación se mostrara un ejemplo de una construcción de un teorema:

theorem rev_rev***: "***rev (rev xs) = xs**"*

El objetivo de este teorema es demostrar que invirtiendo dos veces una lista cualquiera, se obtiene la misma lista de origen.

3.4.6 Ejemplo de uso de métodos para la demostración de un Teorema/Lema

Mediante los siguientes métodos podemos trabajar con diferentes reglas para la demostración de teoremas/lemas:

Método ***erule***: Este método está diseñado para trabajar con reglas de eliminación.

A través de una regla de eliminación podemos llegar a una conclusión a partir de un conjunto de premisas que se cumplen.

Por ejemplo:

$$[\![?P \vee ?Q ;\ ?P \Longrightarrow ?R ;\ ?Q \Longrightarrow ?R]\!] \Longrightarrow ?R$$

Esta regla de eliminación, denominada ***disjE***, nos indica que si se cumple el conjunto de premisas (entre corchetes) podemos concluir dicha conclusión.

Método ***rule***: Este método está diseñado para trabajar con reglas de introducción.

Una regla de introducción nos indica cómo podemos demostrar un conclusión que contiene un símbolo lógico especifico.

Por ejemplo:

$$[\![?P ; ?Q]\!] \Longrightarrow ?P \wedge ?Q$$

Esta regla de introducción, denominada ***conjI***, nos indica que si tenemos una conclusión cómo $?P \wedge ?Q$ es equivalente a tener dos conclusiones $?P$ y $?Q$ que deberán demostrarse por separado para que la conclusión original sea válida.

3.4.7 Ejemplo de demostración de un Teorema/Lema

A continuación se muestran dos ejemplos de lemas triviales donde se utilizan los métodos rule y erule para la demostración de los lemas.

Ejemplo 1:

lemma *conj_rule:* **"**$[\![P ; Q]\!] \Longrightarrow P \wedge (Q \wedge P)$**"**

apply *(rule conjI)*

apply *assumption*

apply *(rule conjI)*

apply *assumption*

apply *assumption*

En primer lugar aplicamos la regla de introducción ***conjI***, con lo cual conseguimos dividir el objetivo en dos sub-objetivos:

$$[\![P ; Q]\!] \Longrightarrow P$$

$$[\![P ; Q]\!] \Longrightarrow Q \wedge P$$

Aplicando la regla ***assumption***, que consiste en probar una conclusión que coincide con alguna de las premisas que tenemos, indicamos a Isabelle que el primer subobjetivo puede demostrarse mediante dicha regla, ya que se puede encontrar un patrón que cumpla este subobjetivo dentro de la regla ***assumption.*** Por lo que nos quedaría por demostrar:

$$[\![P ; Q]\!] \Longrightarrow Q \wedge P$$

Aplicando nuevamente la regla de introducción ***conjI,*** volvemos a obtener dos subobjetivos:

$$[\![P ; Q]\!] \Longrightarrow Q$$

$$[\![P ; Q]\!] \Longrightarrow P$$

Finalmente aplicamos el mismo razonamiento que en el paso 2 para ambos subobjetivos, ya que al ser triviales, aplicando la regla ***assumption*** a ambos subobjetivos, demostraríamos todos los subobjetivos por lo cual el objetivo inicial quedaría demostrado.

Ejemplo 2:

lemma *disj_swap:* **"**$P \vee Q \Longrightarrow Q \vee P$**"**

apply *(erule disjE)*

apply *(rule disjI2)*

apply *assumption*

apply *(rule disjI1)*

apply *assumption*

Asumimos que se cumple $P \vee Q$, por lo que hay que probar que se cumple $Q \vee P$. En primer lugar aplicamos la regla de eliminación ***disjE***, con lo cual conseguimos dividir el objetivo original en dos nuevos sub-objetivos:

$$P \Longrightarrow Q \vee P$$

$$Q \Longrightarrow Q \vee P$$

Aplicando la regla rule ***disjI2***, el primer sub-objetivo puede sustituirse por:

$$P \Longrightarrow P$$

Aplicando la regla ***assumption***, podemos demostrar el primer sub-objetivo dado que es trivial***,*** con lo que solo nos queda el siguiente sub-objetivo***.***

$$Q \Longrightarrow Q \vee P$$

Aplicando un razonamiento similar que en el caso del primer sub-objetivo podemos demostrar que finalmente se cumple el objetivo original $Q \vee P$.

4. Modelización de protocolos criptográficos en Isabelle/HOL

El significado de la palabra “protocolo” indica, que debe determinarse una secuencia de interacciones entre entidades destinada a lograr un determinado objetivo y fin.

Por ello se ha de modelizar cada una de las partes involucradas dentro de un protocolo:

- Definir los tipos de agentes/entidades que interactúan dentro del protocolo.
- Definir el comportamiento y las capacidades de cada agente, tanto los “buenos” agentes como los “intrusos”.
- Definir los tipos de mensajes que soporta el protocolo.
- Definir los distintos tipos de eventos/sucesos que pueden tener lugar dentro del protocolo.
- Definir los distintos tipos de reglas que deberán cumplirse dentro del protocolo para llegar a un determinado objetivo y fin.

A continuación procederemos a explicar cómo podemos modelizar un protocolo criptográfico en Isabelle, apoyándonos en los conceptos mostrados en los anteriores puntos y en el *método inductivo* para la verificación de protocolos criptográficos desarrollado por Larry Paulson (Paulson.L).

4.1 Modelización de los agentes y tipos de mensajes

En todo protocolo criptográfico es de vital importancia modelizar el comportamiento de los agentes y del tipo de mensajes que estos agentes podrán tratar. Podríamos decir que modelizar los agentes y los mensajes es el primer paso que tendríamos que dar para modelizar un protocolo criptográfico.

Para modelizar los agentes hay que crear un tipo de datos llamado ***agent***, en el cual especificaremos los distintos tipos de agentes que intervienen en el protocolo tanto los agentes honestos, como los que podrían tener malas intenciones como por ejemplo el espía.

Una vez que tenemos los agentes modelizados tendríamos qué pararnos a pensar que mensaje o tipos de mensajes se podrían enviar en el protocolo que queramos modelizar, y una vez pensado, declararemos un tipo de datos que especificara lo que pueden contener los mensajes en las conversaciones entre los agentes.

4.2 Modelización del agente espía

El espía es una parte fundamental del sistema y debe ser incorporado en el modelo. El espía es un usuario malicioso que no tiene que seguir el protocolo .El espía suele observar la red y utiliza las claves que él sabe para descifrar mensajes. Así, irá acumulando ***keys*** y "***nonces***". Estas se pueden usar para redactar nuevos mensajes y enviárselo a otra persona haciéndose pasar por alguien que no es, con la intención de poder llegar a sacar más información.

Dos funciones son las que nos permiten modelizar el comportamiento del espía ***analz*** y ***synth***. Cada función asigna un conjunto de mensajes a otro conjunto de mensajes. El conjunto ***analz H*** formaliza lo que el adversario puede aprender de la serie de mensajes ***H***. Las propiedades de cierre de este conjunto son definidas inductivamente. El conjunto de mensajes falsos que un intruso podría inventar a partir de ***H*** es ***synth (analz H)***, donde ***synth H*** formaliza lo que el adversario puede construir a partir del conjunto de mensajes ***H***.

4.3 Modelización de los eventos

Durante la ejecución de un protocolo se produce una secuencia de eventos/sucesos (una traza) cuya finalidad es la de establecer un estado que nos indique en qué punto o situación del protocolo nos podemos encontrar en un momento determinado.

Un ejemplo sencillo de un evento puede ser el envío de un mensaje desde un agente a otro dentro de un protocolo. Otro ejemplo que nos podemos encontrar es la recepción de esos mensajes (si lo queremos hacer más explícito).

Dependiendo de la modelización del protocolo, dado que puede soportar, por ejemplo, tanto el envío y recepción, como solo el envío (la recepción estaría implícita), en Isabelle podemos crear distintos tipos de datos para la especificación de dichos eventos.

Para modelizar el envío de mensajes de un agente ***A*** a otro ***B***, podemos definir el siguiente tipo de datos en Isabelle:

Ejemplo de definición de evento:

datatype *event = Say A B X*

Este evento expresa el intento del agente ***A*** de enviar un mensaje ***X*** al agente ***B***.

4.4 Modelización del protocolo

La modelización del protocolo determinara qué tipos de eventos podrán tener lugar en dicho protocolo y qué reglas deberán de cumplir todos los agentes para llegar a un determinado objetivo y fin

Para ello, en Isabelle, para definir cada paso del protocolo se especifica mediante una regla de introducción de una definición inductiva.

Dicha definición inductiva determinara todos los conjuntos de trazas (cada traza es una secuencia de eventos) posibles que podrá tener lugar en un protocolo.

A continuación vamos a mostrar cómo en Isabelle podemos definir un conjunto inductivo para la construcción del conjunto de todos los números pares.

Ejemplo de definición de conjunto inductivo:

inductive_set even **::** "nat set" **where**

zero **:** **"** 0 $\in$ even**"** **|**

step **:** **"** n $\in$ even $\Longrightarrow$ (Suc (Suc n)) $\in$ even**"**

zero y ***step*** son los nombres que se le dan a las reglas de introducción del conjunto inductivo ***even***, que define ***"nat set"***, es decir, un conjunto natural.

Dado que la ejecución de un protocolo va a estar compuesto de varios pasos (una traza), será necesario definir una lista de eventos que nos permita registrar los distintos tipos de eventos y determinar el orden en el que aparecen.

Ejemplo de definición de conjunto inductivo para la modelización del protocolo de clave pública Needham-Schroeder:

A continuación mostraremos la formalización del protocolo de clave pública:

1. $A \rightarrow B : \{Na, A\}_{Kb}$
2. $B \rightarrow A : \{Na, Nb, B\}_{Ka}$
3. $A \rightarrow B : \{Nb\}_{Kb}$

Este protocolo especifica el intercambio de mensajes entre los agentes **A** y ***B***, cuyos mensajes están encriptados bajo claves públicas K_a y K_b.

En Isabelle, la definición del conjunto inductivo que soporta dicho protocolo es la siguiente:

```
inductive_set ns_public :: "event list set"
  where

    Nil:  "[] ∈ ns_public"

  | Fake: "⟦evsf ∈ ns_public;  X ∈ synth (analz (knows Spy evsf))⟧
           ⟹ Says Spy B X  # evsf ∈ ns_public"

  | NS1:  "⟦evs1 ∈ ns_public;  Nonce NA ∉ used evs1⟧
           ⟹ Says A B (Crypt (pubK B) {|Nonce NA, Agent A|})
                # evs1  ∈  ns_public"

  | NS2:  "⟦evs2 ∈ ns_public;  Nonce NB ∉ used evs2;
            Says A' B (Crypt (pubK B) {|Nonce NA, Agent A|}) ∈ set evs2⟧
           ⟹ Says B A (Crypt (pubK A) {|Nonce NA, Nonce NB, Agent B|})
                # evs2  ∈  ns_public"

  | NS3:  "⟦evs3 ∈ ns_public;
            Says A  B (Crypt (pubK B) {|Nonce NA, Agent A|}) ∈ set evs3;
            Says B' A (Crypt (pubK A) {|Nonce NA, Nonce NB, Agent B|})
               ∈ set evs3⟧
           ⟹ Says A B (Crypt (pubK B) (Nonce NB)) # evs3 ∈ ns_public"
```

Dicho conjunto tiene el nombre de ***ns_public***, que define el conjunto de todas las posibles listas de eventos (trazas) que pueden tener lugar en dicho protocolo. Esta definición es definida por ***"event list set".*** Cada evento que se produzca se añadirá al principio de la lista, por lo que, si se recorre la lista de izquierda a derecha, el primer evento de una ejecución del protocolo será el último elemento de la lista y el último evento será el primer elemento de la lista.

Nil, Fake, NS1, NS2 y ***NS3*** son las reglas de introducción del conjunto inductivo que determinaran, mediante unas condiciones específicas de cada protocolo, si una lista de eventos (una traza) puede pertenecer a dicho conjunto, es decir, si esa secuencia de eventos puede tener lugar en el protocolo de clave pública.

5. Modelización de Fiat-Shamir en Isabelle/HOL

Debido a que en toda especificación de un protocolo criptográfico se debe referir a una teoría sintáctica de mensajes, para el protocolo **Fiat-Shamir** se han definido los siguientes conceptos que participan en el protocolo.

5.1 Modelización de los agentes y tipos de mensajes

Definimos el tipo de datos ***agent*** que nos proporcionara los distintos tipos de agentes que podrán actuar en el protocolo. En el proyecto habrá un ***Victor,*** un ***Spy*** e infinitas ***Peggy***. En este caso, ***Victor*** será el agente cuya tarea será la de verificar la declaración de una agente ***Peggy*** cualquiera, cuya tarea será la de probar que su declaración es verdadera. Por último, se introduce el agente ***Spy*** que podrá ver toda la conversación entre ***Victor*** y ***Peggy*** e intentara hacerse pasar por una agente ***Peggy*** con el propósito de engañar a ***Victor***.

Definición del agente en Isabelle:

datatype agent = Victor | Peggy nat | Spy

A continuación definimos el tipo de datos que especificara lo que puede contener el mensaje entre una comunicación de los agentes. Debido a que en el protocolo de Fiat-Shamir, la agente ***Peggy*** envía únicamente un número al agente ***Victor***, y este envía a ***Peggy*** un valor booleano, la definición en Isabelle de este tipo de datos, quedara de la siguiente forma:

Definición del tipo de mensaje en Isabelle:

datatype msg = Number nat | Bool bool

5.2 Modelización del agente espía

El espía dispone de las siguientes funciones que serán utilizadas para simular su comportamiento dentro del protocolo fiat-shamir.

- **synthSpy :: "agent ⇒event list => (agent × nat × bool × nat) list"**

Mediante esta función el espía reemplaza su nombre de la lista de conversaciones en la que aparece, por el nombre del agente pasado como parámetro. El retorno de dicha función será una lista de tuplas que contendrán la información de cada conversación en la que aparece el espía.

Ejemplo de uso de la función:

synthSpy *(Peggy 3) ⟦* ***Says3*** *Spy Victor (5) 2,* ***Says2*** *Victor Spy (True) 2,* ***Says1*** *Spy Victor (8) 2⟧ ⟹ ⟦ (Peggy 3, 8, True, 5) ⟧*

Componente	**Descripción**
synthSpy	Función que reemplaza el nombre del espía en las conversaciones en las que aparece por el nombre del agente pasado en el primer parámetro.
Agent	Nombre del agente que se utilizara para reemplazar el nombre del espía.
event list	Lista de conversaciones dentro de las cuales aparecerá el espía.
(agent × nat × bool × nat) list	Lista de tuplas que se devuelve que recogen los mensajes enviados dentro de cada conversación en la que está presente el espía, pero cuyo nombre se reemplaza por el del agente pasado en el primer parámetro.

- **synthSpy2 :: "agent ⇒ msg ⇒nat ⇒event list => (agent × nat × bool × nat) list"**

Mediante esta función, utilizada por la función ***synthSpy***, se procede a generar una tupla con la información suministrada por la función ***synthSpy*** que obtiene el tercer mensaje de una conversación y se lo envía a esta función para que busque los siguientes mensajes de dicha conversación y a su vez genere las siguientes tuplas de las demás conversaciones.

Ejemplo de uso de la función:

synthSpy2 *(Peggy 3) 5 2 ⟦**Says2** Victor Spy (True) 2, **Says1** Spy Victor (8) 2⟧ ⟹ ⟦ (Peggy 3, 8, True, 5) ⟧*

Componente	Descripción
synthSpy2	Función que recibe de ***synthSpy*** la información del tercer mensaje de una conversación y busca los siguientes mensajes de dicha conversación para generar una tupla y a su vez procede a buscar las siguientes tuplas de las siguientes conversaciones.
agent	Nombre del agente que se utilizara para reemplazar el nombre del espía.
msg	Información del tercer mensaje de una conversación que envía la función ***synthSpy*** y que es utilizada por la función ***synthSpy2*** para generar una tupla.
nat	Información de la numeración de la conversación a la que pertenece el mensaje enviado en el segundo parámetro por la función ***synthSpy*** y que es utilizada por la función ***synthSpy2*** para buscar los siguientes mensajes de dicha conversación.
event list	Lista de conversaciones que quedan pendientes de procesar y dentro de las cuales aparecerá el espía.
(agent × nat × bool × nat) list	Lista de tuplas que se devuelve que recogen los mensajes enviados dentro de cada conversación en la que está presente el espía, pero cuyo nombre se reemplaza por el del agente pasado en el primer parámetro.

- **synthSpy3 :: "agent ⇒msg ⇒msg ⇒nat⇒ event list => (agent × nat × bool × nat) list"**

Mediante esta función, utilizada por la función ***synthSpy2***, se procede a generar una tupla con la información suministrada por la función ***synthSpy2*** que obtiene el tercer mensaje (suministrado por la función ***synthSpy***) y el segundo mensaje de una conversación y se lo envía a esta función para que busque los siguientes mensajes de dicha conversación y a su vez genere las siguientes tuplas de las demás conversaciones.

Ejemplo de uso de la función:

synthSpy3 *(Peggy 3) 5 True 2 ⟦* ***Says1*** *Spy Victor (8) 2⟧ ⟹ ⟦ (Peggy 3, 8, True, 5) ⟧*

Componente	**Descripción**
synthSpy3	Función que recibe de ***synthSpy2*** la información del segundo y del tercer mensaje de una conversación y busca los siguientes mensajes de dicha conversación para generar una tupla y a su vez procede a buscar las siguientes tuplas de las siguientes conversaciones.
agent	Nombre del agente que se utilizara para reemplazar el nombre del espía.
msg	Información del tercer mensaje de una conversación que envía la función ***synthSpy2*** y que es utilizada por la función ***synthSpy3*** para generar una tupla.
msg	Información del segundo mensaje de una conversación que envía la función ***synthSpy2*** y que es utilizada por la función ***synthSpy3*** para generar una tupla.
nat	Información de la numeración de la conversación a la que pertenecen los mensajes enviados en el primer y segundo parámetro por la función ***synthSpy2*** y que es utilizada por la función ***synthSpy3*** para buscar los siguientes mensajes de dicha conversación.
event list	Lista de conversaciones que quedan

	pendientes de procesar y dentro de las cuales aparecerá el espía.
(agent × nat × bool × nat) list	Lista de tuplas que se devuelve que recogen los mensajes enviados dentro de cada conversación en la que está presente el espía, pero cuyo nombre se reemplaza por el del agente pasado en el primer parámetro.

5.3 Modelización de los eventos

El comportamiento del sistema esta formalizado como un conjunto de trazas de eventos. Hay definidos tres tipos de eventos:

- **Says1 agent agent msg nat**

Este evento se utiliza por primera vez siempre que el agente 1 (primer parámetro) se pone en contacto con el agente 2 (segundo parámetro) en una nueva conversación. Dado que en el protocolo Fiat-Shamir, el que inicia la conversación es o bien Peggy o el Espía, el agente 1 únicamente podrá tener el valor ***Peggy i*** (siendo i un numero) o ***Spy***. El receptor de esta comunicación siempre será Victor por lo que, el agente 2 siempre tendrá el valor ***Victor***.

El mensaje que el agente 1 quiere transmitir al agente 2, se indica en ***msg*** (tercer parámetro) que indicara el tipo de mensaje que se está transmitiendo. Según el protocolo, el primer dato que se transmite es un valor numérico.

Finalmente, el último parámetro, ***nat***, es un numero natural que identifica la conversación entre el agente 1 y el agente 2. Siempre será uno nuevo cada vez que se inicia una nueva conversación del agente 1.

Ejemplo de Evento:

Says1 (Peggy 3) Victor (8) 2

Componente	Descripción
Says1	Función que envía el mensaje msg desde el agente 1 al agente 2 de una nueva conversación.

agent	Agente 1 que envía el mensaje. Solo podrá tener el valor ***Peggy nat*** o ***Spy***.
agent	Agente 2 que recibe el mensaje. Solo podrá tener el valor ***Victor***.
msg	Dato que transmite el agente 1 al agente 2. Solo podrá contener un valor numérico.
nat	Identificador de la conversación.

- **Says2 agent agent msg nat**

Este evento se utiliza como respuesta a la recepción del evento Says1. El agente 1 (primer parámetro) emitirá una respuesta al agente 2 (segundo parámetro). Dado que el agente que siempre responde es Victor, el primer parámetro siempre tendrá el valor Victor. El agente receptor de este envió podrá ser o bien una Peggy específica o bien el Espía. Según el protocolo, el dato de respuesta es un valor booleano. Dado que la conversación es la misma, el último parámetro tendrá el mismo valor que el recibido en el primer evento Says1.

Ejemplo de Evento:

Says2 Victor (Peggy 3) (True) 2

Componente	**Descripción**
Says2	Función que envía el mensaje msg desde el agente 1 al agente 2 de la conversación especificada.
Agent	Agente 1 que envía el mensaje. Solo podrá tener el valor ***Victor***.
Agent	Agente 2 que recibe el mensaje. Solo podrá tener el valor ***Peggy nat*** o ***Spy***.
msg	Dato que transmite el agente 1 al agente 2. Solo podrá contener un valor booleano.
Nat	Identificador de la conversación.

- ***Says3 agent agent msg nat***

Este evento se utiliza como respuesta a la recepción del evento Says2. El agente 1 (primer parámetro) emitirá una respuesta al agente 2 (segundo parámetro). Dado que el agente que emite la respuesta en este evento es siempre o bien una Peggy específica o el Espía, el primer parámetro siempre tendrá el valor ***Peggy i*** (siendo i un numero) o ***Spy***. El agente receptor de este envió siempre será Victor, por lo que el segundo parámetro siempre tendrá el valor **Victor**. Según el protocolo, el dato de respuesta es un valor numérico. Dado que la conversación es la misma, el último parámetro tendrá el mismo valor que el recibido en los eventos Says1 y Says2.

Ejemplo de Evento:

Says3 (Peggy 3) Victor (5) 2

Componente	Descripción
Says3	Función que envía el mensaje msg desde el agente 1 al agente 2 de la conversación especificada.
Agent	Agente 1 que envía el mensaje. Solo podrá tener el valor ***Peggy nat*** o ***Spy***.
Agent	Agente 2 que recibe el mensaje. Solo podrá tener el valor ***Victor***.
msg	Dato que transmite el agente 1 al agente 2. Solo podrá contener un valor numérico.
Nat	Identificador de la conversación.

5.4 Modelización del protocolo

Cada paso del protocolo es especificado por una regla de una definición inductiva. Una traza de eventos (la historia de las conversaciones entre los agentes) viene especificada por el tipo de datos ***event list,*** por lo que declaramos la constante ***fiat_shamir*** como un conjunto de tales trazas (conjunto de historias posibles).

El conjunto inductivo definido en Isabelle para la especificación del protocolo es el siguiente:

inductive_set *fiat_shamir* :: **"event list set"**

A continuación se detalla cada una de las reglas que definen el conjunto inductivo ***fiat_shamir*** y que formalizan el protocolo.

1. **Regla Nil**

Esta regla introduce la traza vacía.

"[] ∈ fiat_shamir"

2. **Regla NS1A**

Esta regla indica que si una específica "***Peggy Z***" (***Z*** es un numero) no ha iniciado ninguna conversación antes, puede iniciar una conversación con el identificador número 1, dado que es la primera conversación que realizara.

*"⟦evs1 ∈ fiat_shamir; X=(R*R)mod(fn (Peggy Z));coprime R (fn (Peggy Z));Says1 (Peggy Z) B M N ∉ set evs1 ⟧ ⟹ Says1 (Peggy Z) Victor (Number X) 1 # evs1 ∈ fiat_shamir"*

3. **Regla NS1B**

Esta regla indica que si el espía no ha iniciado una conversación cuyo identificador es ***N***, puede iniciar una nueva conversación con ese identificador.

"⟦evs1 ∈ fiat_shamir;Says1 Spy Victor M N ∉ set evs1 ⟧ ⟹ Says1 Spy Victor (Number X) N # evs1 ∈ fiat_shamir"

4. **Regla NS2A**

Esta regla indica que si ***Victor*** ha recibido un mensaje del emisor ***P***, en este caso puede contestar con el valor booleano ***True*** al remitente ***P***.

"⟦evs1 ∈ fiat_shamir; Says1 P Victor M1 N ∈ set evs1 ⟧ ⟹ Says2 Victor P (Bool True) N # evs1 ∈ fiat_shamir"

5. **Regla NS2B**

Esta regla indica que si ***Victor*** ha recibido un mensaje del emisor ***P***, en este caso puede contestar con el valor booleano ***False*** al remitente ***P***.

"⟦evs1 ∈ fiat_shamir; Says1 P Victor M1 N ∈ set evs1 ⟧ ⟹ Says2 Victor P (Bool False) N # evs1 ∈ fiat_shamir"

6. **Regla NS3**

Esta regla indica que si el agente ***P*** ha enviado un mensaje y ha recibido la respuesta de ***Víctor***, en este caso con el valor ***True***, puede contestar con el valor numérico ***Y*** a ***Víctor***.

*"⟦evs1 ∈ fiat_shamir; X=(R*R)mod(fn P);Says1 P Victor (Number X) N ∈ set evs1; Says2 Victor P (Bool True) N ∈ set evs1;Y=R*(fs P) mod(fn P) ⟧ ⟹ Says3 P Victor (Number Y) N # evs1 ∈ fiat_shamir"*

7. **Regla NS3A**

Esta regla indica que si el ***espía*** ha enviado un mensaje y ha recibido la respuesta de ***Víctor***, el ***espía*** puede contestar con el valor numérico ***Y*** a ***Víctor***.

"⟦evs1 ∈ fiat_shamir;Says1 Spy Victor (Number X) N ∈ set evs1; Says2 Victor Spy (Bool b) N ∈ set evs1 ⟧ ⟹ Says3 Spy Victor (Number Y) N # evs1 ∈ fiat_shamir"

8. **Regla NS3B**

Esta regla indica que si el agente ***P*** ha enviado un mensaje y ha recibido la respuesta de ***Víctor,*** en este caso con el valor ***False***, puede contestar con el valor numérico ***Y*** a ***Víctor***.

*"⟦evs1 ∈ fiat_shamir; X=(R*R)mod(fn (Peggy Z));Says1 (Peggy Z) Victor (Number X) N ∈ set evs1; Says2 Victor P (Bool False) N ∈ set evs1;Y=R mod(fn P) ⟧ ⟹ Says3 (Peggy Z) Victor (Number Y) N # evs1 ∈ fiat_shamir"*

9. **Regla NS4**

Esta regla indica que si el agente ***P*** ha enviado a ***Víctor*** la última respuesta de la conversación, puede empezar una nueva conversación con él.

*"⟦evs1 ∈ fiat_shamir; ∃M3 . Says3 P Victor M3 N ∈ set evs1; Says3 P Victor M (N+1) ∉ set evs1; X'=(R'*R')mod(fn P)⟧ ⟹ Says1 P Victor (Number X') (N+1) # evs1 ∈ fiat_shamir"*

5.5 Definiciones Auxiliares para enunciar los teoremas

Función numValidas. Esta función calcula el número de conversaciones entre Peggy/spy y Victor que finalizan correctamente, con los números recibidos esperados.

La función tiene la siguiente cabecera:

numValidas:: " (agent × nat × bool × nat) list => nat "

Hace el cálculo correspondiente al protocolo y si obtenemos el resultado esperado sumamos 1 a numValidas.

Ejemplo de uso de la función:

Suponemos que existen los siguientes cuartetos: (***Peggy3 X True Y),*** (***Peggy3 X' False Y')***

La función ***numValidas*** si **b=0** deberá comprobar que **x= y^2 mod n** y si **b=1** deberá comprobar que **x=y2 v mod n.**Si la comprobación tiene éxito ***numValidas*** será 1, en caso contrario será 0.

Función listaSextetos. Esta función busca que existan dos cuartetos con la misma ***P*** y la misma ***X***, pero distintas ***B***,***Y*** y nos devuelve un sexteto de la forma **(P,X,B,Y,B',Y')**

La función tiene la siguiente cabecera:

listaSextetos:: " (agent × nat × bool × nat) list =>(agent × nat × bool × nat × bool × nat) list

Dada una lista de cuartetos de la forma ***(P,X,B,Y)*** la función comprueba que existan dos cuartetos con la misma ***P***,***X*** y diferentes ***B***,***Y*** y en caso de encontrarlo añade el sexteto **(P,X,B,Y,B',Y')** a la lista de sextetos. Para hacer esta función hemos creado una ***funcion aux*** que es la que dada un cuarteto de entrada se encarga de buscar si existe algún cuarteto con la misma ***P*** y ***X*** pero distinto ***B*** y si es así crear el sexteto correspondiente. Esta función es útil para el teorema *special soundness*, donde se trata de probar que si una Peggy comete el error de usar dos veces la misma nonce, su secreto puede ser deducido.

Ejemplo de uso de la función:

Suponemos que existen los siguientes cuartetos: (***Peggy3 44 True 66),*** (***Peggy3 44 False 133),*** (***Peggy8 56 True 82),*** (***Peggy8 56 false 646),*** (***Peggy10 44 True 66)***

La función ***listaSextetos*** creara una lista de sextetos con los siguientes sextetos. (***Peggy3 ,44, True 66, False , 133) y*** (***Peggy8 56 True 82, False , 646)***

Función cuartetos. Esta función dada una lista de eventos devuelve un cuarteto de la forma**(P,X,B,Y).**

La función tiene la siguiente cabecera:

cuartetos :: "event list => (agent × nat × bool × nat) list" and

cuartetos2 :: "agent ⇒ msg1 ⇒nat ⇒event list => (agent × nat × bool × nat) list" and

cuartetos3 :: "agent ⇒msg1 ⇒msg1 ⇒nat⇒ event list => (agent × nat × bool × nat) list" where

Esta función comprueba que existe alguna conversación terminada entre ***P*** o ***Spy*** y **Victor**. Para esto primero comprobamos que hay un evento de la forma: ***Says3 P Victor (Number X) N*** si esto sucede le pasamos a *cuartetos2* esa misma ***P, Number X y la N***, por lo tanto quedaría así: ***cuartetos2 P (Number X) N evs.*** Seguidamente pasamos a buscar otro evento de la forma ***Says2 Victor P' (Bool B) N'*** en este evento hay que comprobar si ***P*** y ***N*** son las mismas que ***P'*** y ***N'***, para saber que estamos en la misma conversación, si esto es así le pasamos a *cuartetos3* lo siguiente: ***cuartetos3 P (Number X) (Bool B) N evs else []*** y por último, buscamos otro evento de la forma ***Says1 P' Victor (Number Y) N'*** y de nuevo comprobamos si la ***P*** y ***N*** son iguales que ***P'*** y ***N'*** y si así sucede ya tenemos un *cuarteto* de la forma ***(P,X,B,Y).***

Ejemplo de uso de la función:

Suponemos que existen los siguientes eventos: ***Says3 (Peggy 1) Victor (5) 1, Says2 Victor (Peggy 1) (True) 1,Says1 (Peggy 1) Victor (20) 1 .***

La función cuartetos comprueba que la conversación está terminada ya que los tres eventos pertenecen a la misma ronda y es correcto en cuanto a la forma que tienen los eventos. Por lo tanto la función *cuartetos* devolverá una lista de cuartetos, con el siguiente cuarteto:

(Peggy 1, 20,True ,5)

Función filtrado. Esta función busca un agente en una lista de eventos y devuelve una lista de eventos en los que únicamente aparezca ese agente.

La función tiene la siguiente cabecera:

filtrado:: "agent ⇒event list ⇒event list"

La función primero busca un evento de la forma ***Says1 P' V M N*** y si ***P***= ***P'*** añade este evento a una nueva lista de eventos y si no sigue buscando otro evento de esa forma. Luego buscamos otro evento de la forma ***Says2 V P' M N*** y si ***P***= ***P'*** añade este evento a la nueva lista de eventos y si no sigue buscando otro evento de esa forma. Por ultimo busca un evento de la forma ***Says3 P' V M N*** y si ***P***= ***P'*** añade este evento a la nueva lista de eventos y si no sigue buscando otro evento de esa forma.

Ejemplo de uso de la función:

Suponemos que existen los siguientes eventos: ***Says3 (Peggy 1) Victor (5) 1, Says2 Victor (Peggy 1) (True) 1, Says1 (Peggy 1) Victor (20) 1.,Says3 (Peggy 9) Victor (5) , Says2 Victor (Peggy 7) (True), Says1 (Peggy 5) Victor (20) 1***

Si buscamos ***peggy 1*** la función filtrar devolverá la siguiente lista de eventos: ***Says1 (Peggy 1) Victor (20) 1 , Says2 Victor (Peggy 1) (True) 1,Says3 (Peggy 1) Victor (5) 1.***

Si buscamos ***peggy 9*** la función filtrar devolverá la siguiente lista de eventos: ***Says3 (Peggy 9) Victor (5) 1***

Aparte de todas las funciones que acabamos de detallar hemos necesitado crearnos un conjunto inductivo llamado ***smallSets***.

Este conjunto inductivo nos permitirá crear conjuntos de naturales cuyos elementos cumplirán las reglas definidas en dicho conjunto inductivo.

La definición del conjunto ***smallSets*** es la siguiente:

```
inductive_set

smallSets :: "nat => nat => nat => nat set set"
```

for N :: nat and K :: nat and V :: nat

where

unit: "⟦ n < N ⟧ ⟹ {n} ∈ smallSets N K V" |

add: "⟦ n < N; S ∈ smallSets N K V; card S < K ⟧

⟹ S ∪ { n } ∈ smallSets N K V"|

prod: "⟦ S ∈ smallSets N K V; Q ∈ S; W ∈ S ; card S < K ⟧

⟹ S ∪ { Q*W mod N } ∈ smallSets N K V"|

inv: "⟦ S ∈ smallSets N K V; x ∈ S ; card S < K; (x*y mod N=1) ⟧

⟹ S ∪ { y mod N } ∈ smallSets N K V"|

uve: "{ V } ∈ smallSets N K V"

A continuación describiremos las reglas definidas en el conjunto inductivo ***smallSets***:

Regla *unit*: Mediante esta regla, podemos crear conjuntos unitarios que pertenezcan al conjunto inductivo.

Regla *add*: Mediante esta regla, podemos crear conjuntos más grandes que pertenezcan a ***smallSets*** a partir de conjuntos más pequeños que pertenezcan a su vez a ***smallSets***.

Regla *prod*: Mediante esta regla, podemos crear conjuntos más grandes que pertenezcan a ***smallSets*** a partir de conjuntos más pequeños que pertenezcan a su vez a ***smallSets*** mediante el modulo del producto de sus elementos.

Regla *inv:* Mediante esta regla, podemos crear conjuntos más grandes que pertenezcan a ***smallSets*** a partir de conjuntos más pequeños que pertenezcan a su vez a ***smallSets*** mediante el modulo del inverso de sus elementos.

Regla *uve:* Mediante esta regla, podemos crear conjuntos unitarios que pertenezcan al conjunto inductivo. Se diferencia de la regla *unit*, en que su elemento se define a través del tercer parámetro, y en la regla *unit*, mediante el primero.

6. Enunciado y demostración de los teoremas relevantes.

6.1 Teorema de completitud

Si ***Peggy*** y ***Víctor*** siguen el protocolo (son honestos) con entrada común ***v***, y entrada privada ***s*** para ***Peggy***, Cuando ***(v,s) ∈ R***, *V* siempre acepta.

∀n . ∀m . ∃t ∈ FiatShamir . numValidas(cuartetos(filtrado(Peggy m) t)) = n

Demostración:

Por inducción sobre ***n***
caso ***n = 0***.
Hay que probar ***∀m : ∃t ∈ FiatShamir . numValidas(cuartetos(filtrado(Peggy m) t)) = 0***, lo cual se cumple trivialmente para la traza **t = [ ].**

caso n > 0.
Fijado un agente honesto ***m***, por *hipótesis de inducción* suponemos la existencia de una traza ***t*** que satisface el teorema. Hay que probar la existencia de una traza ***t' ∈ FiatShamir*** tal que:

numValidas(cuartetos(filtrado(Peggy m) t´)) = n + 1

Elegimos:

t' = Says3 P V Y (n + 1) # Says2 V P T (n + 1) # Says1 P V X (n + 1) # t

donde ***P = Peggy m***, ***V = Victor***, ***X = Number x, x = r2 mod n_m, r*** es cualquier número menor que ***n_m, T = Bool True, Y = Number y, y = rs_m mod n_m, s_m*** es el secreto de ***Peggy m***, y (***n_m, v_m***) es la clave pública de ***Peggy m***.

El resto de la demostración consiste en probar:

1. ***t´ ∈ FiatShamir***. Debería ser posible, utilizando las reglas inductivas *NS3, NS2A* y *NS1B*, y sabiendo que ***t ∈ FiatShamir***.

2. ***numValidas(cuartetos(filtrado(Peggy m) t')) = n + 1***. Al aplicar ***filtrado*** y ***cuartetos*** a ***t'*** , debería aparecer un cuarteto más que los ***n*** de ***cuartetos(filtrado(Peggy m) t).*** Sólo quedaría probar que ese cuarteto pasa el test de ***numValidas***.

6.2 Teorema de la propiedad de corrección especial

Expresa que, si una Peggy ***m*** comete el error de tener dos conversaciones válidas con Victor empleando la misma ***x*** inicial y contestando éste con dos desafíos ***b*** distintos, entonces el espía puede calcular eficientemente una raíz cuadrada ***s*** de la clave pública v_m de Peggy. Es decir, ***s*** cumpliría la misma propiedad que el secreto de Peggy, y a partir de aquí el espía podría suplantar a Peggy cuantas veces quisiera.

$$\forall m . \forall t \in FiatShamir . listaSextetos(cuartetos(filtrado(Peggy\ m)\ t)) = [(Peggy\ m; Number\ x; True; y1; False; y0)] \Longrightarrow \exists s \in smallSets\ n_m\ 10\ v_m . \exists s \in S.\ s^2 v_m \bmod n_m = 1$$

Demostración:

Si la lista de sextetos contiene (***Peggy m; Number x; True; y_1; False; y_0***), esto quiere decir que las conversaciones (***x; 1; y_1***) y (***x; 0; y_0***) son válidas. Por tanto, tenemos:

$$x = y_0^2 \bmod n_m \qquad \text{y} \qquad x = y_1^2\ v_m \bmod n_m$$

Por tanto, $v_m = y_0^2\ y_1^{-2} \bmod n_m$. Por otro lado el $y_0^{-1} \bmod n_m$ es un y_0' unívoco que cumple $y_0 y_0' \bmod n_m = 1$, y que puede ser calculado eficientemente mediante el algoritmo de Euclides extendido en menos de *10* pasos a partir ***de n_m*** e ***y_o***.

Elegimos $s = y_0'\ y_1 \bmod n_m$. Vemos que esta ***s*** cumple la conclusión del teorema:

$$s^2 v_m \bmod n_m = (y_0'\ y_1 \bmod n_m)^2\ v_m \bmod n_m = (\ y_0'^2\ y_1^2\ \ y_0^2\ y_1^{-2}\ ) \bmod n_m = (y_0 y_0' \bmod n_m)^2 = 1$$

6.3 Teorema que implica conocimiento cero del verificador

Expresa que si el espía conoce de antemano el booleano ***b*** que va a enviar Victor como desafío, entonces para cualquier Peggy, el espía puede construir cualquier número menor que ***n*** de conversaciones que Victor da por válidas, siendo ***n*** la clave pública de Peggy.

$$\forall b\ .\ \forall m\ .\ \forall r{<}n_m.\ \exists t \in FiatShamir\ .\ \exists S \in smallSets\ n_m\ 10\ v_m{:}\ \exists x \in S\ .$$
$$Says1\ Spy\ Victor\ (Number\ x)\ 1 \in set\ t$$
$$\wedge\ Says2\ Victor\ (Bool\ b)\ 1 \in set\ t$$
$$\wedge\ Says3\ Spy\ Victor\ (Number\ (r\ mod\ n_m))\ 1 \in set\ t$$
$$\wedge\ numValidas(synthSpy\ (Peggy\ m)\ (filtrado\ Spy\ t)) \geq 1$$

Demostración:
Por casos sobre ***b***.

caso b = false.
Fijamos un agente arbitrario ***m*** y elegimos:

$$t = Says3\ Spy\ Victor\ (Number\ (r\ mod\ n_m))\ 1\#Says2\ Victor\ (Bool\ false)\ 1\#$$
$$Says1\ Spy\ Victor\ (Number\ x)\ 1\#[]$$

donde $x = r^2 mod\ n_m$. El conjunto $S \in smallSets$ se define $S = \{r, x, y\}$,con $y = r\ mod\ n_m$..
En ese caso, Victor comprueba $x = y^2\ mod\ n_m$ y ***numValidas*** devuelve ***1***.

caso b = true.
Fijamos un agente arbitrario ***m*** y elegimos:

$$t = Says3\ Spy\ Victor\ (Number\ (r\ mod\ n_m))\ 1\#Says2\ Victor\ (Bool\ true)\ 1\#$$
$$Says1\ Spy\ Victor\ (Number\ x)\ 1\#[]$$

donde $x = r^2\ v_m\ mod\ n_m$. El conjunto $S \in smallSets$ se define $S = \{r, r^2\ mod\ n_m, v_m, x, y\}$,con $y = r\ mod\ n_m$. En ese caso, Victor comprueba $x = y^2 v_m\ mod\ n_m$ y ***numValidas*** devuelve 1.

7. Conclusión

El objetivo de este proyecto denominado ”VERIFICACIÓN DE SEGURIDAD DE UN PROTOCOLO CRIPTOGRAFICO UTILIZANDO UN ASISTENTE DE DEMOSTRACIÓN” era demostrar que el protocolo criptográfico de conocimiento cero llamado de Fiat-Shamir es un ∑- protocolo, y en consecuencia es en efecto un protocolo de conocimiento cero.

Al realizar este proyecto hemos conseguido completar nuestro objetivo inicial ya que por medio del demostrador llamado Isabelle, hemos llegado a probar que el protocolo criptográfico de conocimiento cero llamado de Fiat-Shamir es un ∑- protocolo.

Para probar que el protocolo criptográfico de conocimiento cero es un ∑- protocolo hemos demostrado los teoremas del apartado 6 de esta memoria (teorema de completitud, teorema de la propiedad de corrección especial y el teorema que implica conocimiento cero del verificador) usando el demostrador Isabelle.

El proyecto nos ha enseñado a ser conscientes de la importante y la utilidad de los asistentes de demostración, en particular del denominado Isabelle. Al principio del proyecto nos costó mucho entender las nociones básicas sobre Isabelle y saber cómo poder empezar, por lo que estuvimos meses buscando información y aprendiendo a usar el programa. En esta primera etapa del proyecto, los avances fueron muy lentos, por la gran dificultad que encontramos a la hora de encontrar información y todo el tiempo que dedicábamos a conocer el programa y a hacernos a él. Una vez adquiridos los conocimientos que necesitábamos para poder empezar, comenzamos a utilizar por fin Isabelle y ya más o menos fuimos capaces de afrontar el reto de demostrar los teoremas.

Durante todo este proyecto ha sido fundamental la ayuda que nos han prestado los dos profesores tanto Ricardo como María Emilia, ya que ellos siempre nos prestaban la ayuda oportuna tanto para corregir alguna cosa como para poder continuar en muchos momentos en los que no sabíamos cómo seguir.

Finalizando este proyecto no se podría llegar a decir que somos unos expertos en Isabelle, pero sí podemos decir que hemos sido capaces de aprender muchísimo y hemos demostrado lo aprendido llegando a demostrar los teoremas que necesitábamos para probar el objetivo de nuestro proyecto.

8. Referencias

Bella.G. *Inductive Verification of Cryptographic Protocols. PhD Thesis, University of Cambridge (2000).*

Cramer.R. *Modular Design of Secure yet Practical Cryptographic. (PhD, Univ. De Amsterdam 1997. ISBN 9074795-64-1.).*

Damgard.I. [En línea] http://www.daimi.au.dk/~ivan/Sigma.pdf.

Feige.U, Fiat.A , Shamir.A. *Zero-knowledge proofs of identity. Journal of Cryptology 1 (2) 77-94, (1988).*

Goldwasser.S, Micali.S , Rackoff.C. *The knowledge complexity of interactive proof-systems .Journal of Computing, 18, pp.186-208, (1989).*

M, Wenzel. The Isabelle/Isar Reference Manual. [En línea] http://isabelle.in.tum.de/doc/isar-ref.pdf.

Nipkow.T. [En línea] https://www4.in.tum.de/~nipkow/LNCS2283/.

Paulson.L. [En línea] http://www.cl.cam.ac.uk/~lp15/papers/Auth/jcs.pdf.

Trappe. W, Washington. L.C. *Introduction to Cryptography with coding Theory. 2nd , ed. Pearson Londres, 2005.*

9. Apéndice: Fichero Isabelle + Isar

A continuación se mostrarán todos los tipos de datos, funciones y axiomas definidos para la demostración de los teoremas mediante el código implementado en Isabelle.

```
header{**}

theory Proyecto
imports Main "~~/src/HOL/Number_Theory/Cong"
begin

datatype  --{*We allow any number of friendly agents*}
  agent = Victor | Peggy nat | Spy


datatype
  msg1 = Number nat | Bool bool


text{*Four new events express the traffic between an agent and his card*}
datatype
  event =  Says1  agent agent msg1 nat
          |Says2  agent agent msg1 nat
          |Says3  agent agent msg1 nat


consts
fv    :: "agent => nat" (*dado un agente te devuelve un valor v*)
fn    :: "agent => nat" (*dado un agente te devuelve un valor n*)
fs    :: "agent => nat" (*dado un agente te devuelve un valor n*)
compute2 :: "nat set => nat => nat => (nat× nat) set "
numRonda :: "nat  => nat"
inverso :: "nat => nat => nat"


axiomatization where

inj_fv : "inj fv" and
inj_fs : "inj fs" and
publicnotzero: "fn (Peggy m) ≠ 0" and
secretPublic : "∀ m. fv (Peggy m)=(inverso (fn (Peggy m)) ((fs (Peggy m))^2))
                                  mod (fn (Peggy m))" and
inversProp : "∀ n x. (x *  inverso n x) mod n = 1" and
inversoExiste : "(x < n) ⟶ (∃ x' < n . x' = inverso n x)"




primrec
(*Esta funcion calcula el numero de conversaciones entre Peggy/spy y  victor que
correctamente,con los numeros recibidos esperados.*)
numValidas :: " (agent × nat × bool × nat) list => nat "
where
    "numValidas [] = 0" |
```

```
51      "numValidas (ev # evs)=
52             (case ev of
53             (P,X,B,Y) => (if(B ∧ (X=(((Y*Y)*(fv P)) mod(fn P))))∨(¬B ∧ (X=(Y*Y)
54                           then (1 + numValidas evs) else numValidas evs))"
55
56  primrec
57  aux:: "agent  => nat =>  bool => nat =>(agent × nat × bool × nat) list =>
58                                        (agent × nat × bool × nat × bool × nat)
59  where
60      "aux P X B Y []= []"
61      "aux P X B Y (ev # evs)=
62             (case ev of
63                   (P',X',B',Y') => (if P=P' ∧ X=X'  ∧ B≠B' then [(P,X,B,Y,B',Y')
64                                    else aux  P X B Y evs))"
65
66  primrec
67  (*Esta funcion busca que existan dos cuartetos con la misma P y la misma  X,
68  pero distintas B,Y
69   y nos devuelve un sexteto de la forma (P,X,B,Y,B',Y') .*)
70  listaSextetos:: " (agent × nat × bool × nat) list =>
71   (agent × nat × bool × nat × bool × nat) list "
72  where
73      "listaSextetos [] = []"
74      "listaSextetos (ev # evs)=
75             (case ev of
76               (P,X,B,Y) => aux P X B Y evs @ listaSextetos evs )"
77
78
79
80
81  fun
82    cuartetos   :: "event list => (agent × nat × bool × nat) list"  and
83    cuartetos2   :: "agent  ⇒ msg1 ⇒nat ⇒event list =>
84                      (agent × nat × bool × nat) list" and
85    cuartetos3   :: "agent  ⇒msg1  ⇒msg1  ⇒nat⇒ event list =>
86                      (agent × nat × bool × nat) list" where
87
88     "cuartetos [] = []"
89     "cuartetos (ev # evs) =
90           (case ev of
91                Says3 P Victor (Number X) N =>  cuartetos2 P (Number X) N evs)"
92    "cuartetos2 P (Number X) N [] = []"
93    "cuartetos2 P (Number X) N (ev # evs) =
94           (case ev of
95              Says2 Victor P' (Bool B) N' => (if P=P' ∧ N=N'
96   then cuartetos3 P (Number X) (Bool B) N evs else []))"
97     "cuartetos3 P (Number X) (Bool B) N [] = []"
98     "cuartetos3 P (Number X) (Bool B) N (ev # evs) =
99           (case ev of
100              Says1 P' Victor (Number Y) N' => (if P=P' ∧ N'=N then
```

```
101                                                (P,Y,B,X)# cuartetos evs else [])
102
103
104 fun
105   synthSpy   :: "agent ⇒event list => (agent × nat × bool × nat) list"  and
106   synthSpy2   :: "agent  ⇒ msg1 ⇒nat ⇒event list =>
107                   (agent × nat × bool × nat) list" and
108   synthSpy3   :: "agent  ⇒msg1  ⇒msg1  ⇒nat⇒ event list =>
109                   (agent × nat × bool × nat) list" where
110
111   "synthSpy K [] = []"|
112   "synthSpy K (ev # evs) =
113         (case ev of
114             Says3 P' Victor (Number Y) N' => if P'=Spy then
115                         synthSpy2 K (Number Y) N' evs else [])"|
116   "synthSpy2 K (Number X) N [] = []"|
117   "synthSpy2 K (Number X) N (ev # evs) =
118         (case ev of
119            Says2 Victor P' (Bool B) N' => (if P'=Spy ∧ N=N' then
120                        synthSpy3 K (Number X) (Bool B) N evs else []))"|
121   "synthSpy3 K  (Number Y) (Bool B) N [] = []"|
122   "synthSpy3 K (Number Y) (Bool B) N (ev # evs) =
123         (case ev of
124           Says1 P Victor (Number X) N' => (if P=Spy ∧ N'=N then
125                         (K,X,B,Y) # synthSpy K evs else []))"
126
127
128 inductive_set
129 smallSets :: "nat => nat => nat => nat set set"
130 for N :: nat and K :: nat and V :: nat
131 where
132 unit: "⟦ n < N  ⟧ ⟹ {n}  ∈ smallSets N K V" |
133 add: "⟦ n < N; S ∈ smallSets N K V; card S < K    ⟧
134       ⟹ S ∪ { n }  ∈ smallSets N K V"|
135 prod: "⟦ S ∈ smallSets N K V; Q ∈ S;  W ∈ S ; card S < K ⟧
136       ⟹ S ∪ { Q*W mod N }  ∈ smallSets N K V"|
137 inv: "⟦ S ∈ smallSets N K V; x ∈ S ; card S < K; (x*y mod N=1) ⟧
138          ⟹ S ∪ { y mod N }  ∈ smallSets N K V"|
139 uve: "{ V } ∈ smallSets N K V"
140
141
142 inductive_set
143 fiat_shamir :: "event list set"
144 where
145 Nil: "[] ∈ fiat_shamir"(*traza vacia*)
146 | NS1A: "⟦evs1 ∈ fiat_shamir; X=(R*R)mod(fn (Peggy Z));coprime R (fn (Peggy Z));
147           Says1 (Peggy Z) B M N ∉ set evs1 ⟧ ⟹
148           Says1 (Peggy Z) Victor (Number X) 1 # evs1 ∈ fiat_shamir"
149 | NS1B: "⟦evs1 ∈ fiat_shamir;Says1 Spy Victor M N ∉ set evs1 ⟧ ⟹
150  Says1 Spy Victor (Number X) N # evs1 ∈ fiat_shamir"
```

```
151 | NS2A: "⟦evs1 ∈ fiat_shamir; Says1 P Victor  M N ∈ set evs1 ⟧ ⟹
152  Says2 Victor P (Bool True) N # evs1 ∈ fiat_shamir"
153 | NS2B: "⟦evs1 ∈ fiat_shamir; Says1 P Victor  M N ∈ set evs1 ⟧ ⟹
154  Says2 Victor P (Bool False) N # evs1 ∈ fiat_shamir"
155 | NS3: "⟦evs1 ∈ fiat_shamir; X=(R*R)mod(fn P);Y=R*(fs P) mod(fn P);
156          Says1 P Victor  (Number X) N ∈ set evs1;
157          Says2 Victor P (Bool True) N ∈ set evs1 ⟧ ⟹
158          Says3  P Victor (Number Y) N # evs1 ∈ fiat_shamir"
159 | NS3b: "⟦evs1 ∈ fiat_shamir; X=(R*R)mod(fn (Peggy Z));
160          Says1 (Peggy Z) Victor  (Number X) N ∈ set evs1;
161          Says2 Victor P (Bool False) N ∈ set evs1;Y=R mod(fn P) ⟧ ⟹
162          Says3   (Peggy Z) Victor (Number Y) N # evs1 ∈ fiat_shamir"
163 | NS3A: "⟦evs1 ∈ fiat_shamir;Says1 Spy Victor  (Number X) N ∈ set evs1;
164          Says2 Victor Spy (Bool b) N ∈ set evs1 ⟧ ⟹
165          Says3  Spy Victor (Number Y) N # evs1 ∈ fiat_shamir"
166 | NS4: "⟦evs1 ∈ fiat_shamir; ∃M3 . Says3  P Victor M3 N ∈ set evs1;
167         Says3  P Victor M (N+1) ∉ set evs1; X'=(R'*R')mod(fn P)⟧ ⟹
168         Says1  P Victor (Number X') (N+1) # evs1 ∈ fiat_shamir"
169
170
171 primrec
172 filtrado:: "agent ⇒event list ⇒event list"
173 where
174 "filtrado P [] =[]"
175 "filtrado P (ev # evs) =
176          (case ev of
177              Says1  P' V M N  ⇒ (if P = P' then ev # filtrado P' evs
178                                  else filtrado P' evs) |
179              Says2  V P' M N  ⇒ (if P = P' then ev # filtrado P' evs
180                                  else filtrado P' evs) |
181              Says3  P' V M N  ⇒ (if P = P' then ev # filtrado P' evs
182                                  else filtrado P' evs))"
183
184 (* lemas auxiliares a demostrar *)
185
186 axiomatization where
187 rondasPrevias1:  "t ∈ fiat_shamir ⟹
188                   numValidas(cuartetos(filtrado (Peggy m) t)) = n ⟹
189                   n > 0 ⟹
190                   ∃ M . Says3 (Peggy m) Victor M n ∈ set t" and
191 rondasPrevias2:  "t ∈ fiat_shamir ⟹
192                   numValidas(cuartetos(filtrado (Peggy m) t)) = n ⟹
193                   n > 0 ⟹
194                   Says3 (Peggy m) Victor M (n+1) ∉ set t" and
195 noRondasPrevias: "t ∈ fiat_shamir ⟹
196                    numValidas (cuartetos (filtrado P t)) = 0 ⟹
197                    Says1 P V M N ∉ set t" and
198 propListaSextetos:
    "t ∈ fiat_shamir ⟹ listaSextetos (cuartetos (filtrado (Peggy m) t))
199                      = [(Peggy m, x, True, y1, False, y0)] ⟹
```

```
200             (Peggy m, x, True, y1) ∈ set (cuartetos (filtrado (Peggy m) evs)) ∧
201             (Peggy m, x, False, y0) ∈ set (cuartetos (filtrado (Peggy m) evs)) ∧
202             x = y1 * y1 * fv (Peggy m) mod fn (Peggy m) ∧ x = y0 * y0 mod fn (Pe
203 (*********************************Teorema 1*************************************
204 (* Completitud: los agentes honrados pasan el test todas las veces *)
205
206 lemma "∀ n. ∀ m . ∃ evs ∈ fiat_shamir . numValidas (
207        cuartetos (filtrado (Peggy m) evs)) = n"
208       (* eliminamos cuantificadores universales *)
209 apply (rule allI)
210 apply (rule allI)
211       (* induccion sobre la longitud de las trazas *)
212 apply (induct_tac n)
213       (* la traza vacia cumple trivialmente el teorema *)
214  apply (rule_tac x = "[]" in bexI)
215  apply simp
216  apply (rule fiat_shamir.Nil)
217       (* Traza no vacía: sustituimos la variable existencial por una traza valic
218 apply simp
219 apply (erule bexE)
220 apply (rule_tac x="Says3 (Peggy m) (Victor) (Number ((fn(Peggy m) - 1)*
221                      fs(Peggy m) mod fn(Peggy m))) (na+1)#
222        Says2 (Victor)  (Peggy m) (Bool True) (na+1)#
223        Says1 (Peggy m) (Victor) (Number ((fn(Peggy m) - 1)*
224                     (fn(Peggy m) - 1) mod fn(Peggy m))) (na+1)#evs" in bexI)
225       (* al simplificar, solo hay que ver que el nuevo cuarteto es válido *)
226 apply clarsimp
227       (* necesitamos la relación entre el secreto de Peggy y el valor público *)
228 apply (insert secretPublic)
229 apply (erule_tac x=m in allE)
230 apply simp
231       (* hay que usar el teorema: (a*b) mod c = (a mod c * b mod c) mod c  *)
232 apply (subgoal_tac "((((fn (Peggy m) - 1) * fs (Peggy m) mod fn (Peggy m) *
233                     ((fn (Peggy m) - 1) * fs (Peggy m) mod fn (Peggy m))) *
234                     (inverso (fn (Peggy m)) ((fs (Peggy m))²) mod fn (Peggy m)))
235                     mod fn (Peggy m)) =
236                     (((((((fn (Peggy m) - 1) * fs (Peggy m) mod fn (Peggy m)) *
237                     ((fn (Peggy m) - 1) * fs (Peggy m) mod fn (Peggy m))))
238                     mod fn (Peggy m)) *
239                     ((inverso (fn (Peggy m)) ((fs (Peggy m))²) mod fn (Peggy m))
240                     mod fn (Peggy m)) )
241                     mod fn (Peggy m))")
242 apply (rotate_tac 2)
243 apply (erule ssubst)
244 defer
245 thm mod_mult_eq
246 apply (rule mod_mult_eq)
247 defer
248 apply simp
249       (* un par de veces mas  el teorema: (a*b) mod c = (a mod c * b mod c) mod
```

```
250 apply (subgoal_tac " (((fn (Peggy m) - 1) * fs (Peggy m) mod fn (Peggy m)) *
251                      ((fn (Peggy m) - 1) * fs (Peggy m) mod fn (Peggy m)))
252                      mod fn (Peggy m)=
253                      (((fn (Peggy m) - 1) * fs (Peggy m))  * ((fn (Peggy m) - 1)
254                      fs (Peggy m)))
255                      mod fn (Peggy m)")
256 apply (rotate_tac 2)
257 apply (erule ssubst)
258 apply (subgoal_tac " (fn (Peggy m) - 1) * fs (Peggy m) * ((fn (Peggy m) - 1) *
259         fs (Peggy m)) mod fn (Peggy m) *
260        (inverso (fn (Peggy m)) ((fs (Peggy m))²) mod fn (Peggy m)) mod fn (Peggy
261        (((fn (Peggy m) - 1) * fs (Peggy m) * ((fn (Peggy m) - 1) * fs (Peggy m))
262        (inverso (fn (Peggy m)) ((fs (Peggy m))²)) mod fn (Peggy m))")
263 apply (rotate_tac 2)
264 apply (erule ssubst)
265 defer
266  apply (rule sym, rule mod_mult_eq)
267  apply (rule sym, rule mod_mult_eq)
268 defer
269       (* ahora hay que usar la propiedad del inverso modulo n  *)
270 apply (insert inversProp)
271 apply (erule_tac x="fn (Peggy m)" in allE)
272 apply (erule_tac x="(fs (Peggy m))²" in allE)
273       (* un poco de asociatividad y conmutatividad *)
274 apply (subgoal_tac "(fn (Peggy m) - 1) * fs (Peggy m) * ((fn (Peggy m) - 1) *
275                     fs (Peggy m))=
276                     (fn (Peggy m) - 1) * (fn (Peggy m) - 1) * (fs (Peggy m))^2")
277 apply (rotate_tac 3)
278 apply (erule ssubst)
279 defer
280  apply simp
281  apply (rule disjI2, rule sym)
282        (* hay que informar a Isabelle de que x^2 = x*x *)
283  apply (rule power2_eq_square)
284 defer
285        (* de nuevo el teorema: (a*b) mod c = (a mod c * b mod c) mod c  *)
286 apply (subgoal_tac "(fn (Peggy m) - 1) * (fn (Peggy m) - 1) *
287                     (fs (Peggy m))² * inverso (fn (Peggy m)) ((fs (Peggy m))²)
288                     mod fn (Peggy m)=
289                     ((fn (Peggy m) - 1) * (fn (Peggy m) - 1) mod fn (Peggy m) *
290                     (((fs (Peggy m))² * inverso (fn (Peggy m)) ((fs (Peggy m))²)
291                     mod fn (Peggy m)))
292                     mod fn (Peggy m)")
293 apply (rotate_tac 3)
294 apply (erule ssubst)
295 apply (rotate_tac 2)
296 apply (erule ssubst)
297         (* esto si lo sabe simplificar *)
298 apply simp
299         (* otro poco de asociatividad *)
```

```
apply (subgoal_tac "(fn (Peggy m) - 1) * (fn (Peggy m) - 1) * (fs (Peggy m))²
                    * inverso (fn (Peggy m)) ((fs (Peggy m))²)=
                    (fn (Peggy m) - 1) * (fn (Peggy m) - 1) * ((fs (Peggy m))²
                    * inverso (fn (Peggy m)) ((fs (Peggy m))²))")
apply  (rotate_tac 3)
apply (erule ssubst)
apply (rule mod_mult_eq)
apply simp
         (* Pasamos a probar que la traza es válida *)
apply  (rotate_tac 3)
apply  (rotate_tac 3)
apply (thin_tac ?x)
apply (thin_tac ?x)
         (* Vamos mensaje por mensaje. Primero Says1 *)
apply (subgoal_tac "Says1 (Peggy m)Victor (Number ((fn (Peggy m) - 1) *
                 (fn (Peggy m) - 1) mod fn (Peggy m))) (na + 1) # evs ∈ fiat_sha
defer
         (* Distinguimos si es la primera ronda o no *)
apply (case_tac na)
         (* es la primera *)
 apply simp
(*
NS1A: "⟦evs1 ∈ fiat_shamir; X=(R*R)mod(fn (Peggy Z));coprime R (fn (Peggy Z));
          Says1 (Peggy Z) B M N ∉ set evs1 ⟧ ⟹
          Says1 (Peggy Z) Victor (Number X) 1 # evs1 ∈ fiat_shamir"
*)

 apply (rule_tac R="(fn (Peggy m) - 1)" in fiat_shamir.NS1A, simp, simp)
 thm coprime_minus_one_nat
 apply (rule coprime_minus_one_nat)
 apply (rule publicnotzero)
 apply (rule noRondasPrevias, simp, simp)
      (* Ahora el caso en que no es la primera ronda *)
 apply (rule_tac R'="fn (Peggy m) - 1" in  NS4)
 apply assumption
 apply (rule rondasPrevias1, simp, simp, simp)
 apply (rule rondasPrevias2, simp_all)
      (* Ahora vamos a por el mensaje Says2 *)
apply (subgoal_tac " Says2 Victor (Peggy m) (Bool True) (na + 1) #
                    Says1 (Peggy m)Victor (Number ((fn (Peggy m) - 1) *
                   (fn (Peggy m) - 1) mod fn (Peggy m))) (na + 1) # evs ∈ fiat_
defer
      (* Aqui no hay que hacer distincion de casos. Simplemente aplicar NS2A *)
(*
NS2A: "⟦evs1 ∈ fiat_shamir; Says1 P Victor  M N ∈ set evs1 ⟧ ⟹
 Says2 Victor P (Bool True) N # evs1 ∈ fiat_shamir"
*)
apply (rule_tac M="(Number ((fn (Peggy m) - 1) *
                   (fn (Peggy m) - 1) mod fn (Peggy m)))" in fiat_shamir.NS2A)
apply simp
```

```
apply simp
        (* Por último, el mensaje Says3 *)
(*
| NS3: "⟦evs1 ∈ fiat_shamir; X=(R*R)mod(fn P);Says1 P Victor  (Number X) N ∈ set
 Says2 Victor P (Bool True) N ∈ set evs1;Y=R*(fs P) mod(fn P) ⟧ ⟹
 Says3  P Victor (Number Y) N # evs1 ∈ fiat_shamir"
*)
apply (rule_tac X="(fn (Peggy m) - 1) * (fn (Peggy m) - 1) mod fn (Peggy m)"
and
                R="(fn (Peggy m) - 1)"  in fiat_shamir.NS3)
apply  simp_all
done


(* Special Soundness Property: Si una Peggy comete el error de usar la
misma nonce dos veces y emite respuestas ante dos desafios distintos, su
secreto queda comprometido *)

axiomatization where
numerosProtocolo:
"(Peggy m, x, b, y0) ∈ set (cuartetos (filtrado (Peggy m) t)) ⟹
      y0 < (fn (Peggy m))"
(*********************************Teorema 2*************************************

lemma "∀ m . ∀ evs ∈ fiat_shamir .
       listaSextetos (cuartetos (filtrado (Peggy m) evs))=
       [((Peggy m),x,True, y1,False,y0)] ⟶
       (∃ S ∈ smallSets  (fn (Peggy m)) 10  (fv (Peggy m)) .
       ∃ s ∈ S . s * s * fv (Peggy m) mod (fn (Peggy m)) = 1)"

apply (rule allI)
apply (rule ballI)
apply (rule impI)

apply (subgoal_tac "listaSextetos (cuartetos (filtrado (Peggy m) evs))=
                    [((Peggy m),x,True, y1,False,y0)] ⟶
                    (Peggy m,x,True,y1) ∈ set (cuartetos (filtrado (Peggy m) evs
                    (Peggy m,x,False,y0)∈ set (cuartetos (filtrado (Peggy m) evs
                    x=y1*y1*fv (Peggy m) mod (fn (Peggy m)) ∧
                    x=y0*y0 mod (fn (Peggy m))")

apply (subgoal_tac "y0 < fn (Peggy m) --> (∃ y0' < fn (Peggy m). y0' =
                    inverso (fn (Peggy m)) y0)")
defer
apply (rule inversoExiste)
apply simp
defer
      (* nos dedicamos al objetivo principal *)

apply (drule mp)
```

```
398 apply assumption
399 apply (drule mp)
400 apply (elim conjE)
401 apply (rule numerosProtocolo, simp)
402 apply (elim conjE)
403
404 apply (erule exE)
405 apply (elim conjE)
406 apply (thin_tac ?x)
407 apply (thin_tac ?x)
408 apply (thin_tac "(?x,?y,False,?z) ∈ ?w")
409
410
411 apply (rule_tac x="{y0'}  ∪                      (* y'0 *)
412                   {y1}  ∪                        (* y1 *)
413                   {y0' * y1  mod fn (Peggy m)}" (* s = y0' * y1 *)
414                   in bexI)
415
416 apply (rule_tac x="y0' * y1  mod fn (Peggy m)" in bexI)
417 defer
418 apply simp
419               (*  Probamos que el conjunto {y0',y1,y0' * y1 mod fn (Peggy m)}
420                   es un small set    *)
421
422 apply (subgoal_tac "{y0'} ∈ smallSets (fn (Peggy m)) 10 (fv (Peggy m)) ")
423 defer
424 apply (rule smallSets.unit)
425 apply  assumption
426 defer
427 defer
428 apply (subgoal_tac "{y0'}∪{y1} ∈ smallSets (fn (Peggy m)) 10 (fv (Peggy m)) ")
429 defer
430 apply (rule_tac S="{y0'}"   in  smallSets.add)
431 apply (rule numerosProtocolo, simp)
432
433 apply simp
434 apply simp
435 defer
436 defer
437
438
439 apply (rule_tac S="{y0'} ∪ {y1}" and Q="y0'" and W="y1"
440       in  smallSets.prod)
441 apply  assumption
442 apply  simp
443 apply  simp
444 apply  simp
445 defer
446      (* Ahora probamos la propiedad aritmética de la s calculada en el small set
447 apply (thin_tac ?x)
```

```
448 apply (insert inversProp)
449 apply (erule_tac x="fn (Peggy m)" in allE)
450 apply (erule_tac x="y0" in allE)
451 apply (rotate_tac 2)
452 apply (thin_tac ?x)
453 (*Metemos la condicion v = v mod n*)
454 apply (subgoal_tac "fv (Peggy m) < fn (Peggy m)  ⟹
455  fv (Peggy m) mod fn (Peggy m)=fv (Peggy m)  ")
456 
457 defer
458 apply (rule mod_less)
459 apply simp
460 defer
461 apply (subgoal_tac " fv (Peggy m) = fv (Peggy m) mod fn (Peggy m) ")
462 apply (rotate_tac 5)
463 apply (erule ssubst)
464 defer
465 apply simp
466 apply (insert secretPublic)
467 apply (erule_tac x=m in allE)
468 apply simp
469 defer
470 (*Intentamos juntar y1*y1*v*)
471 apply (subgoal_tac "y0' * y1 mod fn (Peggy m) * (y0' * y1 mod fn (Peggy m)) *
472        (fv (Peggy m) mod fn (Peggy m)) mod fn (Peggy m)=
473       (y0' * y1  mod fn (Peggy m) * (y0' * y1 mod fn (Peggy m))) mod fn (Peggy m
474        (fv (Peggy m) mod fn (Peggy m)) mod
475        fn (Peggy m) ")
476 defer
477 apply (rule mod_mult_left_eq)
478 defer
479 apply (rotate_tac 4)
480 apply (erule ssubst)
481 (*Propiedad del modulo*)
482 apply (subgoal_tac "y0' * y1 mod fn (Peggy m) *
483                     (y0' * y1 mod fn (Peggy m)) mod fn (Peggy m)=
484                     (y0' * y1) * (y0' * y1) mod fn (Peggy m)")
485 defer
486 apply (rule sym,rule mod_mult_eq)
487 defer
488 apply (rotate_tac 6)
489 apply (erule ssubst)
490 (*Propiedad del modulo*)
491 apply (subgoal_tac "y0' * y1 * (y0' * y1) mod fn (Peggy m) *
492                     (fv (Peggy m) mod fn (Peggy m)) mod fn (Peggy m)=
493                     y0' * y1 * (y0' * y1) * fv (Peggy m) mod fn (Peggy m)")
494 defer
495 apply (rule sym,rule mod_mult_eq)
496 defer
497 apply (rotate_tac 6)
```

```
498 apply (erule ssubst)
499
500 apply (subgoal_tac "y0' * y1 * (y0' * y1)= y1 * y1* y0' * y0'")
501 defer
502 apply simp
503 defer
504 apply (rotate_tac 6)
505 apply (erule ssubst)
506 apply (subgoal_tac "y1 * y1 * y0' * y0' *fv (Peggy m)=
507                     y1 * y1 * fv (Peggy m) * y0' * y0'")
508 defer
509 apply simp
510 defer
511 apply (rotate_tac 6)
512 apply (erule ssubst)
513
514 apply (subgoal_tac "y1 * y1 * fv (Peggy m) * y0' * y0' =
515                     y1 * y1 * fv (Peggy m) * (y0' * y0')")
516 defer
517 apply simp
518 defer
519 apply (rotate_tac 6)
520 apply (erule ssubst)
521 (*Propiedad del modulo*)
522 apply (subgoal_tac "y1 * y1 * fv (Peggy m) * (y0' * y0') mod fn (Peggy m)=
523                     y1 * y1 * fv (Peggy m) mod fn (Peggy m) *
524                     (y0' * y0') mod fn (Peggy m)")
525 defer
526 apply (rule mod_mult_left_eq)
527 defer
528 apply (rotate_tac 6)
529 apply (erule ssubst)
530 apply (subgoal_tac " y1 * y1 * fv (Peggy m) mod fn (Peggy m)= x")
531 defer
532 apply simp
533 defer
534 apply (rotate_tac 6)
535 apply (erule ssubst)
536 apply (rotate_tac 4)
537 apply (thin_tac ?x)
538 apply (erule ssubst)
539 (*Propiedad del modulo*)
540 apply (subgoal_tac " y0 * y0 * fv (Peggy m) mod fn (Peggy m) *
541                     (y0' * y0') mod fn (Peggy m) =
542                      y0 * y0 * fv (Peggy m) * (y0' * y0') mod fn (Peggy m)")
543 defer
544
545 apply (rule sym,rule mod_mult_left_eq)
546 defer
547 apply (rotate_tac 5)
```

```
548 (*Propiedad del modulo*)
549 apply (subgoal_tac "y0 * y0 mod fn (Peggy m) * (y0' * y0') mod fn (Peggy m)=
550 y0 * y0 *(y0' * y0') mod fn (Peggy m)
551 ")
552 defer
553 apply (rule sym,rule mod_mult_left_eq)
554 defer
555 apply (rotate_tac 5)
556 apply (erule ssubst)
557 apply (subgoal_tac "y0 * y0 * (y0' * y0')=y0 * y0'* (y0 * y0')")
558 defer
559 apply simp
560 defer
561 apply (rotate_tac 5)
562 apply (erule ssubst)
563
564 apply (subgoal_tac " y0 * y0' * (y0 * y0') mod fn (Peggy m)=
565 y0 * y0' mod fn (Peggy m) * (y0 * y0') mod fn (Peggy m)
566 ")
567 defer
568 apply (rule mod_mult_left_eq)
569 defer
570 apply (rotate_tac 5)
571 apply (erule ssubst)
572 apply (rotate_tac 2)
573 apply simp
574 (*terminamos la demostracion del subgoal*)
575 apply (rotate_tac 2)
576 apply (thin_tac ?x)
577 apply (thin_tac ?x)
578 (*Axioma*)
579 apply (rule propListaSextetos, simp, simp)
580
581 done
582
583 (* Special Honest-Verifier Zero-Knowledge: Si el desafío es conocido de
584 antemano, el espia puede simular conversaciones indistinguibles de las
585 de un agente honesto *)
586 (*********************************Teorema 3*************************************
587 lemma "∀ b. ∀ m . ∀ r < fn (Peggy m) . ∃ evs ∈ fiat_shamir .
588       ∃ S ∈ smallSets  (fn (Peggy m)) 10  (fv (Peggy m)). ∃ x ∈ S .
589       Says1 Spy Victor (Number x) 1 ∈ set evs ∧
590       Says2 Victor Spy (Bool b) 1 ∈ set evs ∧
591       Says3 Spy Victor (Number (r mod fn (Peggy m))) 1 ∈ set evs ∧
592       numValidas (synthSpy (Peggy m) (filtrado Spy evs)) ≥ 1"
593 apply (rule allI)
594 apply (rule allI)
595 apply (rule allI)
596 apply (rule impI)
597 apply (case_tac b)
```

```
598         (* Caso b=True *)
599 apply (rule_tac x="Says3 Spy Victor (Number (r mod fn (Peggy m))) 1 #
600                    Says2 Victor Spy (Bool b) 1 #
601                    Says1 Spy Victor (Number (r* r * fv (Peggy m) mod fn (Peggy m
602                    []" in bexI)
603 apply (rule_tac x="{r}∪                                        (* r *)
604                     {r * r mod fn (Peggy m)}∪                   (* r^2  mod Nm *)
605                     {fv (Peggy m)}∪                             (* Vm *)
606                     {(r * r * fv (Peggy m)) mod fn (Peggy m)}∪ (* x *)
607                     {r  mod fn (Peggy m)}"                     (* y *)
608                     in bexI)
609 apply (rule_tac x="(r * r * fv (Peggy m)) mod fn (Peggy m)" in bexI)
610        (* primero las cuentas aritméticas *)
611 apply simp
612        (* la x esta en el small set *)
613 apply simp
614        (* Ahora intentamos probar el small set *)
615 apply (subgoal_tac "{r} ∈ smallSets (fn (Peggy m)) 10 (fv (Peggy m)) ")
616 defer
617 apply (rule smallSets.unit)
618 apply  assumption
619 defer
620 defer
621 apply (subgoal_tac "{r}∪{r mod fn (Peggy m)} ∈
622                     smallSets (fn (Peggy m)) 10 (fv (Peggy m)) ")
623 defer
624 apply simp
625 defer
626 defer
627 apply (subgoal_tac "{r}∪{r mod fn (Peggy m)}∪{fv (Peggy m) } ∈
628        smallSets (fn (Peggy m)) 10 (fv (Peggy m)) ")
629 defer
630 (*
631 add: "⟦ n < N; S ∈ smallSets N K V; card S < K ; n  ∉ S    ⟧
632       ⟹ S ∪ { n }  ∈ smallSets N K V"|
633 *)
634
635 apply (rule_tac S="{r}∪{r mod fn (Peggy m)}"   in  smallSets.add)
636         (* la v de Peggy siempre es menor que su n *)
637 apply (insert secretPublic)
638 apply (erule_tac x=m in allE)
639 apply simp
640 apply  assumption
641 apply  simp
642 defer
643 defer
644
645 apply (subgoal_tac "{r}∪{r mod fn (Peggy m)}∪{fv (Peggy m) }∪
646             {r * r mod fn (Peggy m)} ∈ smallSets (fn (Peggy m)) 10 (fv (Peggy m)
647 defer
```

```
(*
prod: "⟦ S ∈ smallSets N K V; n1 ∈ S;  n2 ∈ S ; card S < K ⟧
      ⟹ S ∪ { n1*n2 mod N }  ∈ smallSets N K V"|
*)
apply (rule_tac S="{r} ∪ {r mod fn (Peggy m)} ∪ {fv (Peggy m)}" and Q="r" and
W="r"
      in  smallSets.prod)
apply  assumption
apply  simp
apply  simp
apply  simp
defer
defer

apply (subgoal_tac "{r}∪{r mod fn (Peggy m)}∪{fv (Peggy m) }∪
                    {r * r mod fn (Peggy m)}∪
                    {(r * r mod fn (Peggy m) * fv (Peggy m) ) mod fn (Peggy m)}
                    ∈ smallSets (fn (Peggy m)) 10 (fv (Peggy m)) ")
defer
apply (rule_tac S="{r} ∪ {r mod fn (Peggy m)} ∪ {fv (Peggy m)}
                   ∪ {r * r mod fn (Peggy m)}" and
       Q="r * r mod fn (Peggy m)" and W="fv (Peggy m)"   in  smallSets.prod)
apply  simp
apply  simp
apply  simp
apply  simp
defer
defer
     (* ahora aplicamos la conmutatividad de los conjuntos *)
apply (subgoal_tac "{r} ∪ {r mod fn (Peggy m)} ∪ {fv (Peggy m)} ∪
                    {r * r mod fn (Peggy m)} ∪
       {r * r mod fn (Peggy m) * fv (Peggy m) mod fn (Peggy m)} =
        {r} ∪ {r * r mod fn (Peggy m)} ∪ {fv (Peggy m)} ∪
       {r * r * fv (Peggy m) mod fn (Peggy m)} ∪
       {r mod fn (Peggy m)}")
apply simp
apply (subgoal_tac "r * r mod fn (Peggy m) * fv (Peggy m) mod fn (Peggy m)=
                   r * r * fv (Peggy m) mod fn (Peggy m)")
apply  force
thm mod_mult_left_eq
apply (rule sym, rule mod_mult_left_eq)


   (* Seguimos en el caso b=True. Probamos que esa traza pertenece a fiat_shamir
apply (subgoal_tac "
       [Says1 Spy Victor (Number (r* r * fv (Peggy m) mod fn (Peggy m))) 1]
        ∈ fiat_shamir ")
defer
(*
| NS1B: "⟦evs1 ∈ fiat_shamir;Says1 Spy Victor M N ∉ set evs1 ⟧ ⟹
```

```
697  Says1 Spy Victor (Number X) N # evs1 ∈ fiat_shamir"
698 *)
699
700 apply (rule fiat_shamir.NS1B)
701 apply (rule fiat_shamir.Nil)
702 apply simp
703 defer
704
705       (* Ahora vamos a por el mensaje Says2 *)
706 apply (subgoal_tac "Says2 Victor Spy (Bool True) 1 #
707                  Says1 Spy Victor (Number (r* r * fv (Peggy m) mod fn (Peggy m))
708                  # [] ∈ fiat_shamir ")
709 defer
710       (* Aqui no hay que hacer distincion de casos. Simplemente aplicar NS2A *)
711 (*
712 NS2A: "⟦evs1 ∈ fiat_shamir; Says1 P Victor  M N ∈ set evs1 ⟧ ⟹
713  Says2 Victor P (Bool True) N # evs1 ∈ fiat_shamir"
714 *)
715 apply (rule_tac
716            M="(Number (r* r * fv (Peggy m) mod fn (Peggy m)))" in
               fiat_shamir.NS2A)
717 apply assumption
718 apply simp
719 defer
720
721         (* Por último, el mensaje Says3 *)
722 (*
723 | NS3A: "⟦evs1 ∈ fiat_shamir;Says1 Spy Victor  (Number X) N ∈ set evs1;
724  Says2 Victor Spy (Bool b) N ∈ set evs1 ⟧ ⟹
725  Says3  Spy Victor (Number Y) N # evs1 ∈ fiat_shamir"
726 *)
727 apply (rule_tac b="True" in  fiat_shamir.NS3A)
728 apply  simp
729 apply  simp
730 apply  simp
731
732         (* fin del caso b=True *)
733         (* caso b=False *)
734
735 apply (rule_tac x="Says3 Spy Victor (Number (r mod fn (Peggy m))) 1 #
736                    Says2 Victor Spy (Bool False) 1 #
737                    Says1 Spy Victor (Number (r* r mod fn (Peggy m))) 1 #
738                    []" in bexI)
739 apply (rule_tac x="{r}  ∪                                    (* r *)
740                     {r  mod fn (Peggy m)}  ∪                (* y *)
741                     {(r * r) mod fn (Peggy m)}"             (* x *)
742
743                     in bexI)
744 apply (rule_tac x="(r * r ) mod fn (Peggy m)" in bexI)
745                (* primero las cuentas aritmeticas*)
```

```
746 apply simp
747                (* la x esta en el small set *)
748 apply simp
749 defer          (*posponemos la elim del conj smallSet*)
750 apply simp
751 (*Probamos que esa traza pertenece a fiat_shamir*)
752 apply (subgoal_tac "[Says1 Spy Victor (Number (r* r mod fn (Peggy m))) 1]
753                     ∈ fiat_shamir ")
754 defer
755 (*
756 | NS1B: "⟦evs1 ∈ fiat_shamir;Says1 Spy Victor M N ∉ set evs1 ⟧ ⟹
757  Says1 Spy Victor (Number X) N # evs1 ∈ fiat_shamir"
758 *)
759
760 apply (rule fiat_shamir.NS1B)
761 apply (rule fiat_shamir.Nil)
762 apply simp
763 defer
764
765       (* Ahora vamos a por el mensaje Says2 *)
766 apply (subgoal_tac "Says2 Victor Spy (Bool b) 1 #
767                     Says1 Spy Victor (Number (r* r mod fn (Peggy m))) 1 #
768                     [] ∈ fiat_shamir ")
769 defer
770 apply simp
771       (* Aqui no hay que hacer distincion de casos. Simplemente aplicar NS2A *)
772 (*
773 NS2A: "⟦evs1 ∈ fiat_shamir; Says1 P Victor  M N ∈ set evs1 ⟧ ⟹
774  Says2 Victor P (Bool False) N # evs1 ∈ fiat_shamir"
775 *)
776 apply (rule_tac M="(Number (r* r mod fn (Peggy m)))" in fiat_shamir.NS2B)
777 apply assumption
778 apply simp
779 defer
780
781        (* Por último, el mensaje Says3 *)
782 (*
783 | NS3A: "⟦evs1 ∈ fiat_shamir;Says1 Spy Victor  (Number X) N ∈ set evs1;
784  Says2 Victor Spy (Bool b) N ∈ set evs1 ⟧ ⟹
785  Says3  Spy Victor (Number Y) N # evs1 ∈ fiat_shamir"
786 *)
787
788 apply (rule_tac b="False" in  fiat_shamir.NS3A)
789 apply  simp
790 apply  simp
791 apply  simp
792
793     (* finalmente probamos que es un small set *)
794
795 apply (subgoal_tac "{r} ∈ smallSets (fn (Peggy m)) 10 (fv (Peggy m)) ")
```

```
796 defer
797 apply (rule smallSets.unit)
798 apply  assumption
799 
800 (*
801 add: "⟦ n < N; S ∈ smallSets N K V; card S < K ; n  ∉ S    ⟧
802       ⟹ S ∪ { n }  ∈ smallSets N K V"|
803 *)
804 apply (subgoal_tac "{r}∪{r mod fn (Peggy m)}
805                     ∈ smallSets (fn (Peggy m)) 10 (fv (Peggy m)) ")
806 defer
807 
808 apply (rule_tac S="{r}" in  smallSets.add)
809 apply  simp
810 apply  simp
811 apply simp
812 
813 (*
814 prod: "⟦  S ∈ smallSets N K V ; Q ∈ S ;  W ∈ S ; card S < K ⟧
815       ⟹ S ∪ { Q * W mod N}  ∈ smallSets N K V"|
816 *)
817 
818 apply (rule_tac S="{r} ∪ {r mod fn (Peggy m)}" and Q="r" and W="r"  in
    smallSets.prod)
819 apply  simp_all
820 done
821 
822 
823 end
824 
825 
826 
827 
```